2nd Edition

Basics of Python Programming

Learn Python in 30 days (Beginners approach)

Dr. Pratiyush Guleria

bpb

Basics of Python Programming

Learn Python in 30 days (Beginners approach)

2nd Edition

Dr. Pratiyush Guleria

www.bpbonline.com

Second Revised and Updated Edition 2024

First Edition 2020

Copyright © BPB Publications, India

ISBN: 978-93-55516-404

All Rights No part of this publication may be reproduced, distributed or transmitted in any form or by any means or stored in a database or retrieval system, without the prior written permission of the publisher with the exception to the program listings which may be entered, stored and executed in a computer system, but they can not be reproduced by the means of publication, photocopy, recording, or by any electronic and mechanical means.

LIMITS OF LIABILITY AND DISCLAIMER OF WARRANTY

The information contained in this book is true to correct and the best of author’s and publisher’s knowledge. The author has made every effort to ensure the accuracy of these publications, but publisher cannot be held responsible for any loss or damage arising from any information in this book.

All trademarks referred to in the book are acknowledged as properties of their respective owners but BPB Publications cannot guarantee the accuracy of this information.

To View Complete
BPB Publications Catalogue
Scan the QR Code:

www.bpbonline.com

Dedicated to

Almighty God

My beloved parents

and

My beloved family

About the Author

Dr. Pratiyush Guleria has a Ph.D. in Computer Science. He has done an M.Tech in Computer Science and is a gold medalist at Himachal Pradesh University, Shimla, India. He has a consistent track record in academics throughout his career. He has also qualified for the State Level Eligibility Test (SLET). Dr. Pratiyush Guleria has more than 15+ years of experience in the IT industry and academics. He has research papers published in various peer-reviewed international journals and conferences. He has been a member of the Technical Program Committee and a reviewer for journals and international conferences. His research interests include Data Mining, Machine Learning, and Web Technologies.

About the Reviewers

a passionate Data Enthusiast with a Postgraduate degree in Artificial Intelligence from Great Lakes, embarked on his IT journey with where he honed his skills in SAP BODS, SQL, PYTHON, and Excel, catering to US-based retail clients. Later, at Impact he delved into a diverse tech stack, including R, Python, SQL, AWS, GCP, GUROBI, and more, refining his interpersonal skills.

During his tenure at Jyant led teams in delivering a spectrum of products, from price optimization to movie recommendations and Credit Scoring Models, employing a versatile toolkit encompassing Python, R, Advance SQL, Superset, Airflow, Git, Azure, Pyspark, and others.

Currently, as a leader at Jyant spearheads the development of two pivotal models: an attrition model and an innovative chatbot powered by AWS LEX. His expertise extends to LLm, Rag, and ReAct Prompt integration, alongside utilizing SQLAlchemy to enhance the chatbot's capabilities.

Throughout his journey, Jyant has consistently demonstrated a knack for addressing multifaceted challenges, from crafting Regression models for promotion optimization to wielding Seaborn and Matplotlib for comprehensive EDA, and even applying CNN and RESNET50 for image attribute tagging, NLP for data extraction, creating sophisticated Movie Recommendation Engines using Lightfm Neural Network models, as well

as crafting Price Optimization and Credit Scoring Models using intricate bank and retail data.

Peyush Kumar is an experienced IT professional, with a distinguished career that spans over 25 years. Graduating with a B.Tech in Computer Science from Punjabi University, his journey has been marked by impactful contributions to major tech players, including and This includes employment in global settings while residing for an extended period in the USA. Beyond showcasing prowess as a skilled programmer, Peyush proved himself as a proficient manager, steering software development teams towards success.

Peyush’s enthusiasm extends beyond coding; he is a passionate educator. His teaching legacy is witnessed in the diverse array of students, spanning from high school enthusiasts to seasoned graduates, who have benefitted from his insightful guidance.

Residing in Noida with his family, Peyush balances his love for technology with an unwavering commitment to family values.

Acknowledgement

I want to express my deepest gratitude to my family and friends, especially my parents, for their unwavering support and encouragement throughout this book's writing.

I am also grateful to BPB Publications for their guidance and expertise in bringing this book to fruition. It was a long journey of revising this book, with valuable participation and collaboration of reviewers, technical experts, and editors.

I would also like to acknowledge the valuable contributions of my colleagues and co-workers during many years working in the tech industry, who have taught me so much and provided valuable feedback on my work.

Finally, I would like to thank all the readers who have taken an interest in my book and for their support in making it a reality. Your encouragement has been invaluable.

Preface

The author is optimistic that students who want to complete a thorough using Python will find relief in this work. This book provides a range of real-world examples, conceptual challenges, and methodically worked-out solutions to help readers understand programming ideas through clear explanations and examples. Additionally, it covers every topic necessary for students to gain a thorough understanding of the fundamentals of Python. This book is a great resource for anyone with programming knowledge and a great place for newbies to start.

The abilities needed to understand the fundamentals of Python at a beginner's level are taken into mind when writing this book. The book presents real-world Python examples in an approachable style that makes it easy for students to learn and comprehend.

To err is human, to forgive divine is a proverb. Although the book is written with honesty and sincerity, I hope that its flaws will be overlooked. However, the author is receptive to any form of helpful critiques and recommendations for additional development. All insightful recommendations are welcomed, and the author will do their utmost to include them in worthwhile additions to this work in future versions.

Chapter 1: Introduction to Python Variables, Datatypes and Operators - In this chapter, learners will understand Python software, its installation, and writing programs in IDLE. Going through this chapter, we will cover the

fundamentals of variable declaration, datatypes, and so on. Apart from that, Python operators are illustrated with the help of suitable examples.

Chapter 2: Conditions and Loops - In this chapter, learners will be able to understand the fundamentals of control statements and implement the logic using if conditions and loops wherever it is necessary. Apart from that, readers will be able to learn the difference between break and continue statements. Learners will also grasp the concepts of the control statements, range() function with the help of examples. The concepts of conditions, loops, range() function, break and continue statements are discussed and illustrated with the help of suitable examples.

Chapter 3: Arrays and Functions - Arrays are the variables having similar data types. Arrays in Python contain values corresponding to the same data types, whereas lists, which are somewhat similar to arrays discussed in the next chapter, contain values corresponding to different data types. Arrays can contain more than one value at a time, and indexing is used to refer to array elements. A function in Python is a block of a series of statements that carry out a single or several actions. Code is easier to read, comprehend, and maintain when it is divided into usable, modular chunks because of the use of functions. The ability to reuse code is one benefit of utilizing functions.

Chapter 4: Lists, Tuples, Iterators Generators, and Sets - In this chapter, we will discuss the concept of lists, tuples, iterators, and generators. Although all of these are data structures or constructs that are used to keep, iterate through, and modify collections of data, they each have distinctive characteristics.

Chapter 5: Dictionaries and Modules - In this chapter, we will learn about dictionaries and modules. A dictionary is a commonly used data structure that returns data as key-value pairs. Dictionaries use key-value pairs to store and retrieve data efficiently. Python uses modules to split code into individual files, which makes it easier to organize and reuse. Import statement is used for implementing modules.

Chapter 6: File Handling and Databases - This chapter will cover the topics related to file handling and databases. File handling in Python enables one to work with files using the built-in functions and libraries. In databases, we will discuss the concept of data definition language (DDL) and data manipulation language (DML) commands for database connections.

Chapter 7: Object-Oriented Programming - Python supports object-oriented programming (OOP) concepts. This chapter will cover the OOP concepts like classes, objects, inheritance, overriding, and so on. A class is a template for creating objects and is a collection of data members, and methods. An object is an instance of a class. In classes, methods that can perform actions on the object using dot notation are defined within a class.

Chapter 8: Regular Expressions, Date and Time - Regular expressions are used to identify patterns in a sequence of strings. To work with the regular expressions in Python, we need to import the re module. The regular expressions help us in searching, matching and manipulating strings based on specific patterns. The datetime module in Python is widely utilized for manipulating dates and times. The current date and time can be obtained using datetime.now() function. You may get the current date with date.today() function.

Chapter 9: Exception Handling - Exception is an error that occurs during the execution of the program, and Exception Handling provides a user-friendly interface to handle the exception and prompts a message to the user stating the reason for such unexpected exceptions, like dividing a number by zero.

Chapter 10: NumPy and TKinter - By going through this chapter, learners will acquire the knowledge of Numpy and perform operations on arrays using Numpy. Apart from that, you will be able to work with a GUI-based environment of Python known as Tkinter.

Appendix: Practice Exercises with Solutions - Learners will be presented with practical activities in the Appendix that include fill-in-the-blank questions, true/false questions, and more. A reader who gains an overview of Python essentials through the appendix will also be better equipped to face technical interviews and prepare for entrance-level examinations.

Code Bundle and Coloured Images

Please follow the link to download the

Code Bundle and the Coloured Images of the book:

https://rebrand.ly/e0c1e2

The code bundle for the book is also hosted on GitHub at In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

Did you know that BPB offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.bpbonline.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at :

business@bpbonline.com for more details.

At you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at business@bpbonline.com with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a

general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Table of Contents

1. Introduction to Python Variables, Datatypes and Operators

Introduction

Structure

Objectives

Python installation

Exploring Python

Example 1.1: Program to print Hello World

Variable declaration and getting user input in Python

Example 1.2: Example of user input

Example 1.3: Addition of two numbers

Example 1.4: Write a program to swap two values

Python datatypes

Example 1.5: Example of integer, float, string datatypes, and type conversion

Operators in Python

Example 1.6: Arithmetic operators in Python

Relational operators

Example 1.7: Relational operators in Python

Logical, bitwise and assignment operators

Example 1.8 Example of logical, bitwise, left shift, right shift and assignment operators

Identity operators

Example 1.9: Example of is and isnot identity operator

Membership operators

Example 1.10: Using the membership operator

Precedence of operators

type() function in Python

Example 1.11: Write a program for the type() function

Conclusion

Points to remember

Exercises

2. Conditions and Loops

Introduction

Structure

Objectives

If condition

Example 2.1: Example of if condition

Example 2.2: Example of if condition using and, or operators

Example 2.3: Example of checking conditions using and, or operators

Nested-if condition

Example 2.4: Example of nested-if conditions to find the greatest of three numbers

Loops

Example 2.5: Example of a while loop

Example 2.6: Sum of first five numbers using the while loop

Example 2.7: Demonstrating break statement in a while loop

Example 2.8: Example for finding a palindrome number using a while loop

Example 2.9: Multiplication of a number using a while loop

Example 2.10: Example of for loop

Example 2.11: Example of the list and for loop

Example 2.12: Print the numbers using range() function in for loop

Example 2.13: Write a program to display odd numbers using for loop

Example 2.14: Write a program using for loop with arrays

Example 2.15: Find the factorial of a number using for loop

Example 2.16: Program to find the sum of all numbers stored in a list using for loop

Example 2.17: Finding prime number taking input from the user

Break and continue statement

Example 2.18: Write a program to display the prime numbers from 1 to 20 using for loop

Conclusion

Questions

Points to remember

3. Arrays and Functions

Introduction

Structure

Objectives

Arrays

Example 3.1: Write a program for arrays using append() function

Example 3.2: Example of membership operator in

Example 3.3: Example of in operator and if condition

Example 3.4: Example of len() function in array

Example 3.5: Display values from 0 to 9 using for loop

Example 3.6: Display values from 0 to 9 using arrays and append() function

Functions

Example 3.7: Passing parameters in a function

Local and global variables in functions

Example 3.8: Local and global variable

Example 3.9: Write a program for a variable having global scope

Example 3.10: Write a calculator program using the concept of functions

Example 3.11: Function returning the absolute value of a number

Example 3.12: Write a program to sort array elements

Example 3.13: Write a program to find the square of a number using functions

Lambda function

Example 3.14: Adding numbers using the lambda function

Conclusion

Points to remember

Exercises

4. Lists, Tuples, Iterators Generators, and Sets

Introduction

Structure

Objectives

Lists

Example 4.1: Example of using the append() function in lists

Slicing

Example 4.2: Example of slicing a list and for storing multiple data items

Example 4.3: Example of Slicing using double colon for list sequences

Tuples

Example 4.4: Example for concatenation of tuples

Example 4.5: Nested tuples

Python iterators

Example 4.6: Write a program to print a list of courses using iterators

Example 4.7: Write a program using iterators through loops

Python generators

Example 4.8: Write a program for generators

Example 4.9: Simplify the Example 4.8 using loops

Sets

Example 4.10: Example of a set and its operations

Conclusion

Points to remember

Exercises

5. Dictionaries and Modules

Introduction

Structure

Objectives

Dictionaries in Python

Example 5.1: Write a program for the dictionary

Example 5.2: Accessing values in a dictionary using index and arrays

Example 5.3: Dictionary example for addition and deletion of key-value pairs

Nested dictionary

Example 5.4: Example of nested dictionary

Example 5.5: Counting the number of digits, lowercase, uppercase, and vowels in a sentence

pop() method in a dictionary

Example 5.6: Example of pop() method in dictionary

Modules

Example 5.7: Using modules

Example 5.8: Example of using dictionary using modules

Example 5.9: Example of importing mathematical functions in a module

Conclusion

Points to remember

Exercises

6. File Handling and Databases

Introduction

Structure

Objectives

File and modes used in file handling

Example 6.1: Example for reading a file

Example 6.2: Example for reading every line of the file

Example 6.3: Write data into file using w mode

Example 6.4: Write a program to illustrate append mode

Databases

Data definition language

Data query language

Data manipulation language

Data control language

Transaction control language

Example 6.5: Write a program for database and table creation

Example 6.6: Program to insert a record into a table

Example 6.7: To read records from the table and display them

Example 6.8: Write a program to update the data

Example 6.9: Write a program to delete the data

Conclusion

Points to remember

Exercises

7. Object-Oriented Programming

Introduction

Structure

Objectives

Classes and objects

Example 7.1: Writing a program for class and objects

Example 7.2: Writing a program for functions in a class

Inheritance

Example 7.3: Writing a program for inheritance

Overriding

Example 7.4: Writing a program for overriding

Example 7.5: Write a program for inheritance and function overriding

Encapsulation

Example 7.6: Write a program for encapsulation restricting access to the private variables

Example 7.7: Write the aforementioned example using public access specifier

Conclusion

Points to remember

Exercises

8. Regular Expressions, Date and Time

Introduction

Regular expression for pattern matching

Example 8.1: Write a program for substring function using regular expression

Example 8.2: Regular expression program to search string

Example 8.3: Write a program to check email ID pattern using the function

Example 8.4: Write a program for validating string values only

Example 8.5: Write a program for validating numerical values only

Example 8.6: Regular expression to validate the mobile number digits

Date and time functions

Example 8.7: Write a program to display time

Example 8.8: Write a program to display datetime using now function

Example 8.9: Write a program to obtain the calendar month using calendar module

Conclusion

Points to remember

Exercises

9. Exception Handling

Introduction

Objectives

Exception

Example 9.1: Program of exception error occurred due to inappropriate values

Example 9.2: Program of exception error raised due to different data types

Example 9.3: Program of IOError exception

Example 9.4: Program of ModuleNotFound exception

Example 9.5: Write a program for division by zero exception

Example 9.6: Example of assert statement for exception handling

Conclusion

Points to remember

Exercises

10. NumPy and TKinter

Introduction

Structure

Objectives

1-D array operations using NumPy

Example 10.1: Write a program to create a simple 1D array from a Python list

Example 10.2: Finding sum, mean, max, and min functions using NumPy

Matrix addition and multiplication using NumPy

Example 10.3: Write a program for matrix addition and multiplication using NumPy

Fundamentals of Tkinter main application window

Example 10.4: Example of Tkinter application window

Label and button widget in Tkinter

Example 10.5: Create a label and button widget in Tkinter

Button click event and input from user in Tkinter

Example 10.6: Example of button click event

Example 10.7: Example of taking input from a user using the text field

Example 10.8: Write a program for performing arithmetic operations

Conclusion

Points to remember

Exercises

Appendix: Practice Exercises with Solutions

Sample programs

Output prediction

Multiple choice questions

Fill in the blanks

True/False

Index

CHAPTER 1

Introduction to Python Variables, Datatypes and Operators

Introduction

Python is an interpreted and object-oriented high-level programming language. It was designed by Guido Van Rossum and released in 1991. The latest version of Python is Python 3.12.2.

Python itself is mainly written and implemented in the C programming language for flexibility and effectiveness. Python can also be translated into binary code like Java. Python is used in Mathematics and Machine Learning. Python has had tremendous growth across a wide range of industries, including web development, data science, AI, scientific computing, and so on. Since Python is an interpreted language, running your code does not require compiling it first. Python's line-by-line editing and execution capabilities make it ideal for quick development and experimentation. Because Python is dynamically typed, you do not need to explicitly declare variable types.

The type of a variable is decided by the interpreter at runtime, providing flexibility. With the aid of libraries like Tkinter, desktop applications can be made with Python.

Lists, tuples, and dictionaries are the three basic data structures used in Python to store and manage collections of data. Besides that, topics like modules, object-oriented programming features, databases, file handling, and Numpy are addressed in this book with the help of simple and illustrative examples.

Structure

In this chapter, we will cover the following topics:

Know about Python installation

Writing a program in IDLE shell

Variable declaration

Data types

Operators

type() function

Objectives

In this chapter, learners will come to know about Python software, its installation and writing programs in IDLE. Going through this chapter, we will cover the fundamentals of variable declaration, datatypes and so on. Apart from it, Python operators are illustrated with the help of suitable examples.

Python installation

Python is an open-source software and can be downloaded from its official website, as shown in Figure The most recent versions of Python are available for download at

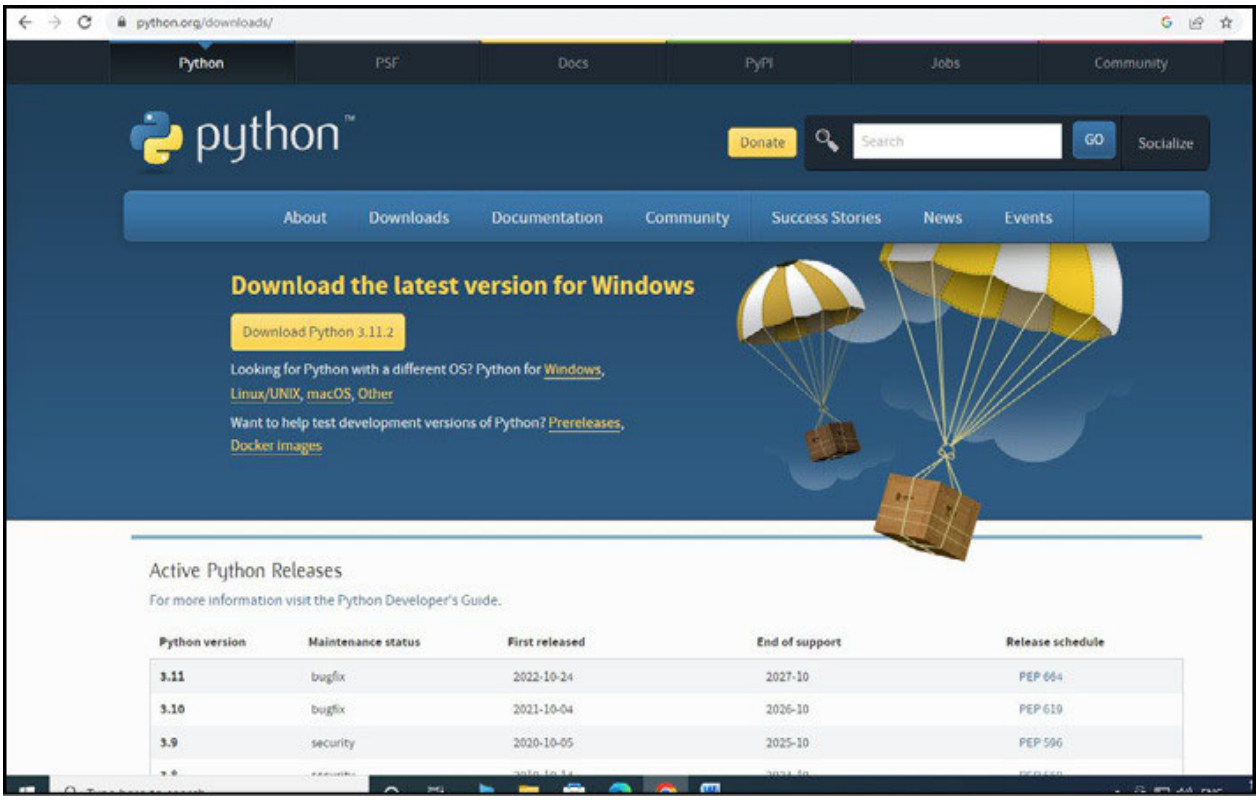


Figure 1.1: Download the Python software

After downloading the software, execute it as shown in Figure

Figure 1.2: Execution of the Python software

After executing the Python software, the setup of the software starts its installation progress and finally displays the completion message of the installation, as shown in Figure

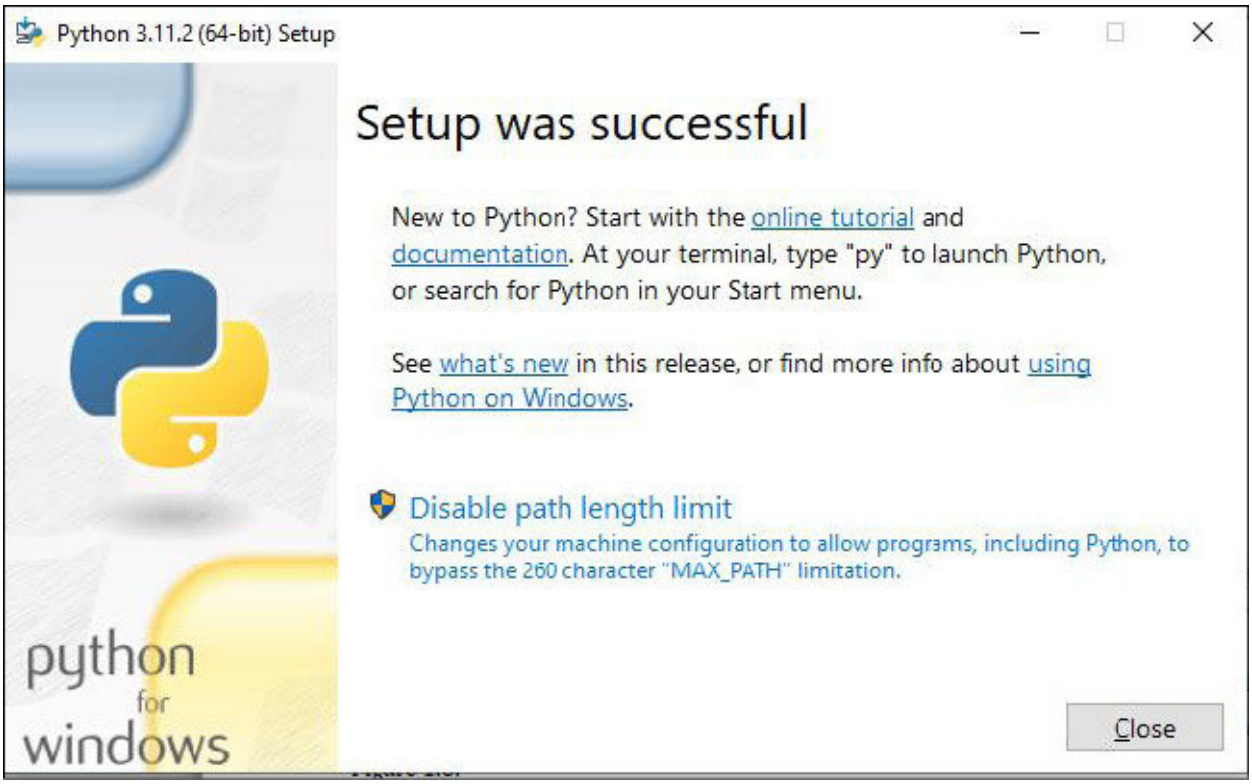

Python 3.11.2 (64-bit) Setup
Setup was successful
New to Python? Start with the online tutorial and documentation. At your terminal, type "py" to launch Python, or search for Python in your Start menu.
See what's new in this release, or find more info about using Python on Windows.
Disable path length limit
Changes your machine configuration to allow programs, including Python, to bypass the 260 character "MAX_PATH" limitation.
python for windows
Close

Figure 1.3: Setup installed successfully

After the Python setup is completed successfully, the Python IDLE 3.11 environment for writing programs is shown in Figure 1.4 and Figure The IDLE is known as the integrated development learning environment. It is a Graphical user interface for writing Python programs. Take a look at the following figure:

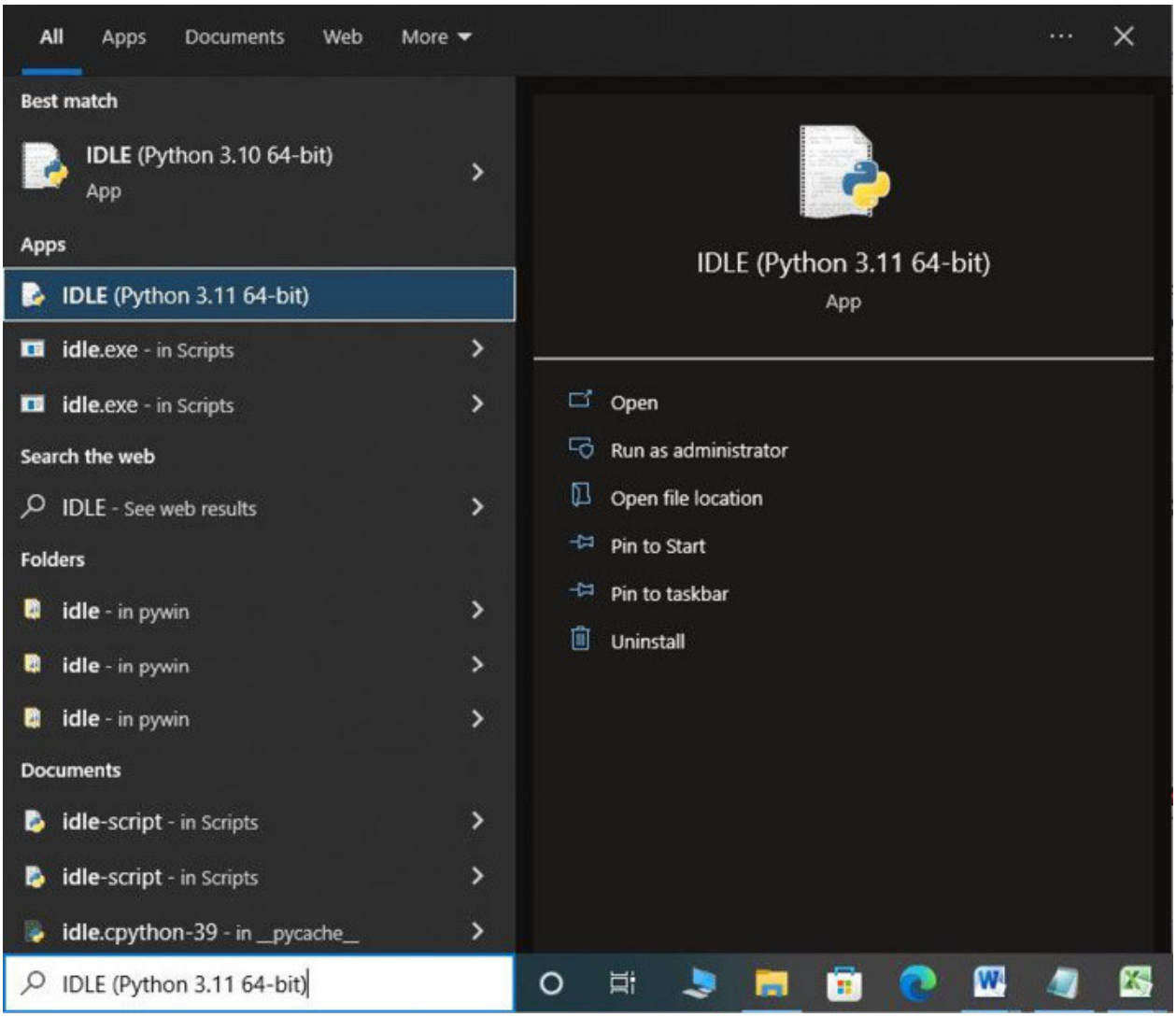


Figure 1.4: Opening IDLE (Python 3.11)

IDLE Python 3.11 Shell is illustrated in the following figure:

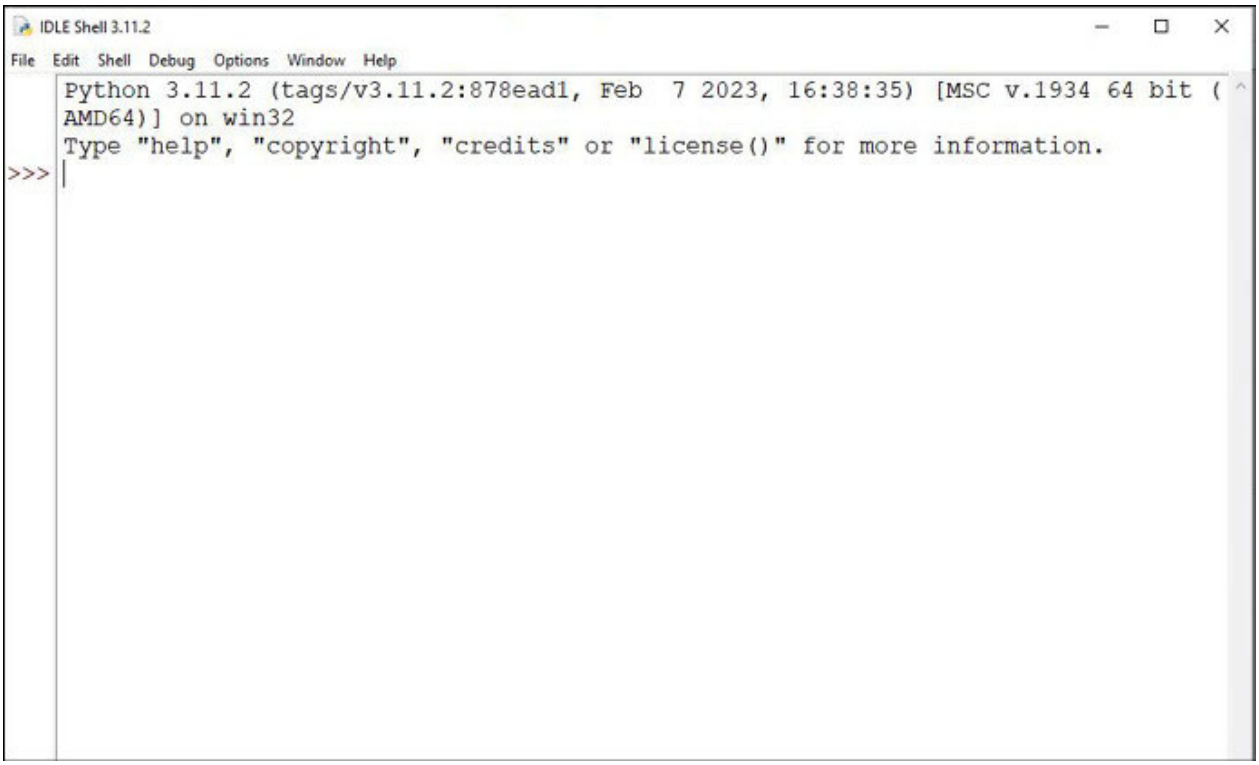

IDLE Shell 3.11.2
File Edit Shell Debug Options Window Help
Python 3.11.2 (tags/v3.11.2:878ead1, Feb 7 2023, 16:38:35) [MSC v.1934 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>

Figure 1.5: IDLE Python 3.11 Shell

Among the widely used tools in the Python development ecosystem are PyCharm, Anaconda, and Visual Studio Code (VS each with a distinct function.

Microsoft created VS Code, a small, open-source code editor. Python is among the many programming languages that it supports. It can be downloaded from a website: https://code.visualstudio.com/ and on opening the software, it looks as shown in Figure

Figure 1.6: VS code welcome window

Specifically created for Python development, PyCharm is an integrated development environment that is robust and feature-rich. Excellent support for Python is offered by PyCharm, including handling packages and virtual environments. Take a look at the PyCharm Figure

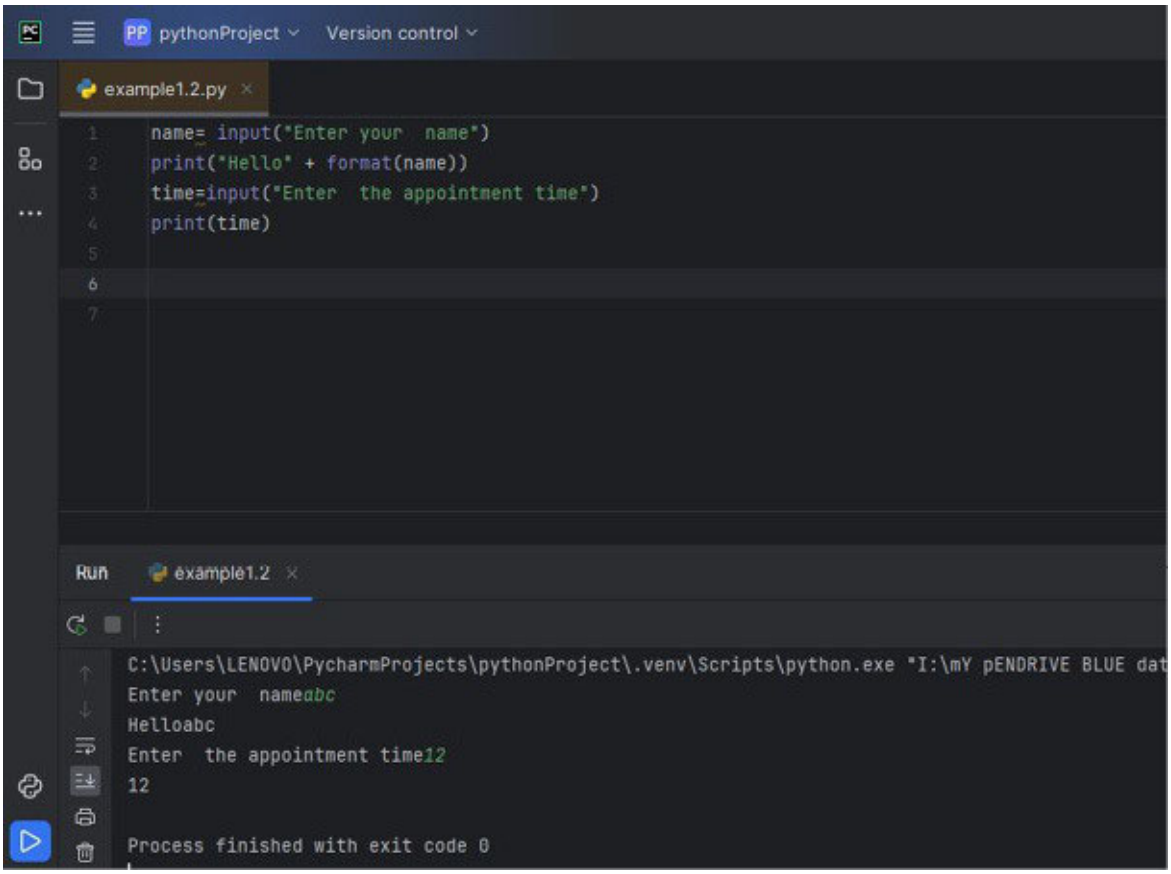


Figure 1.7: PyCharm coding environment

For advanced learning in Python, we need to download the Anaconda software. Python and additional scientific computing programs for data science and machine learning are distributed under the Anaconda namespace. It has instruments for handling packages and environments. First, download the Anaconda Software from a website: the Anaconda software is opened, it appears as Figure

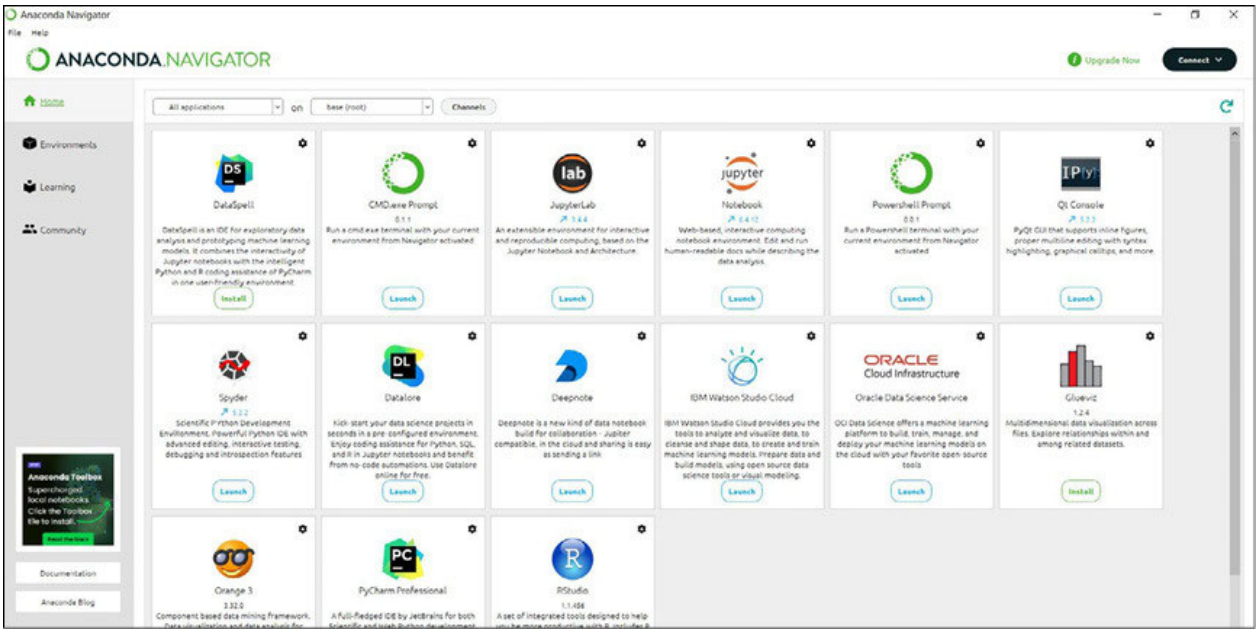


Figure 1.8: Anaconda Navigator environment

The Anaconda software comes with Python software along with built-in libraries of Python. This software consists of two popular IDE integrated development environments which are Spyder and Jupyter Notebook, shown in Figure 1.9 and Figure Interactive coding and output visualization are hallmarks of Jupyter notebooks. Each cell's output is retrieved along with the code in Jupyter, which divides the code into cells. For exploratory analysis and model construction, it is perfect because it supports interactive development. The extension of a Jupyter code file is (.ipynb). Take a look at the following figure:

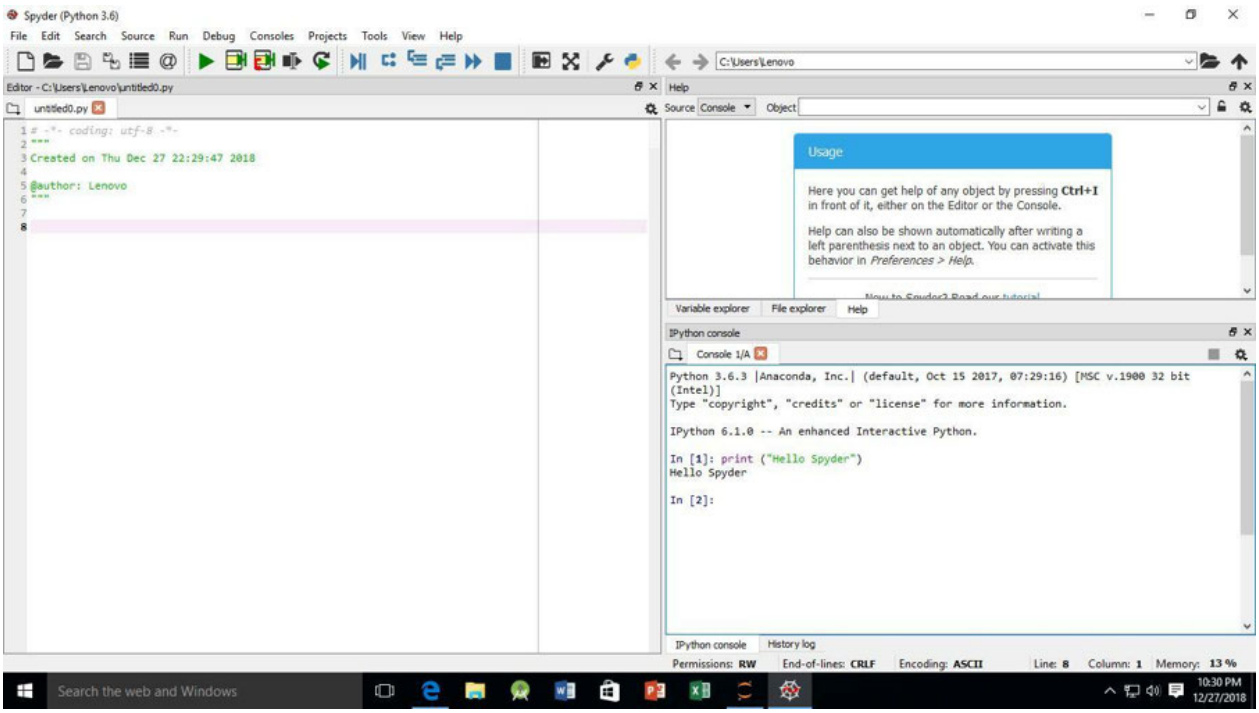

Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor - C:\Users\Lenovo\untitled0.py
untitled0.py
-*- coding: utf-8 -*-
"""
Created on Thu Dec 27 22:29:47 2018
@author: Lenovo
"""
Help
Source Console Object
Usage
Here you can get help of any object by pressing Ctrl+I in front of it, either on the Editor or the Console.
Help can also be shown automatically after writing a left parenthesis next to an object. You can activate this behavior in Preferences > Help.
Variable explorer File explorer Help
IPython console
Console 1/A
Python 3.6.3 |Anaconda, Inc.| (default, Oct 15 2017, 07:29:16) [MSC v.1900 32 bit (Intel)]
Type "copyright", "credits" or "license" for more information.
IPython 6.1.0 -- An enhanced Interactive Python.
In [1]: print ("Hello Spyder")
Hello Spyder
In [2]:
IPython console History log
Permissions: RW End-of-lines: CRLF Encoding: ASCII Line: 8 Column: 1 Memory: 13 %
Search the web and Windows
10:30 PM
12/27/2018

Figure 1.9: Spyder environment window

The Jupyter Environment Window is shown in the following figure:

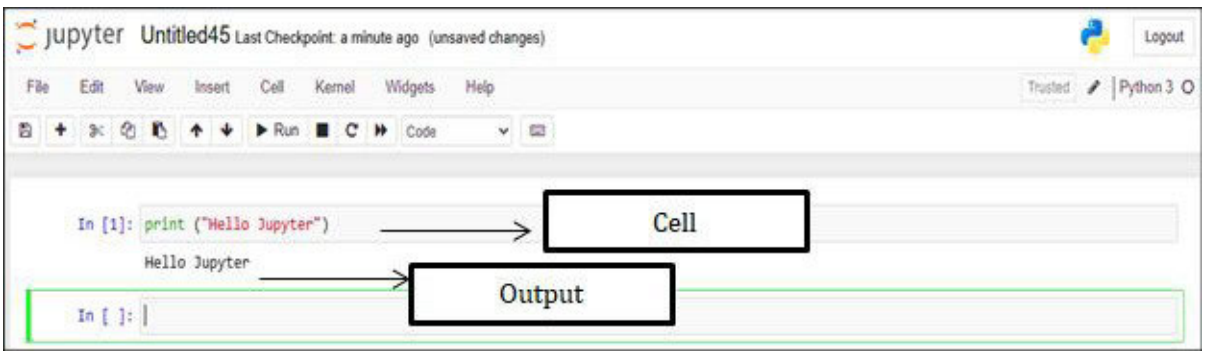


Figure 1.10: Jupyter environment window

Exploring Python

Python can be used in the following ways:

To create web Web applications can be developed in Python with the help of Flask and Django modules.

Connect to database systems.

To handle big Python can be easily integrated with Big Data Frameworks.

Machine learning Python has preinstalled libraries for machine learning, big data, Artificial Intelligence, Natural Language Processing, and so on.

Predictive trends can be developed in Python.

Note: The extension of Python is .py. Python uses indentation as its method of grouping statements. Therefore, in Python, while writing programs, always take care of Indentation. If an indentation error occurs in conditions or loops, the error will prompt an unexpected indent.

The steps to run the Python programs on IDLE Shell are as follows:

Python programs can be executed through Python Shell or IDLE, as shown in Figure 1.11 and Figure Python uses indentation to indicate a block of code.

Create a simple program in IDLE, as shown in Figure

The Python print statement is often used to output variables.

The following figure displays a Python program on IDLE:

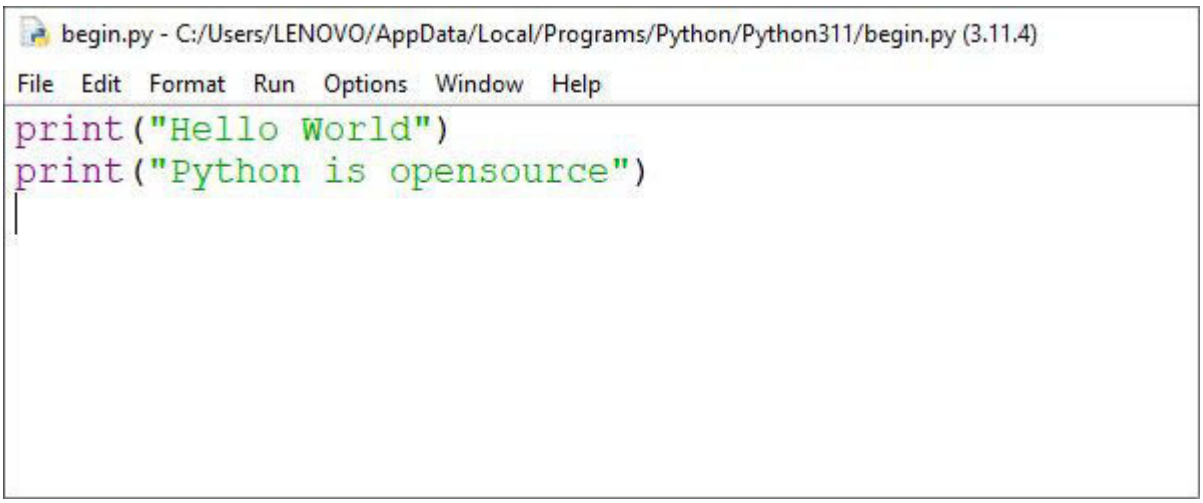

begin.py - C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/begin.py (3.11.4)
File Edit Format Run Options Window Help
print("Hello World")
print("Python is opensource")

Figure 1.11: Window displaying Python program on IDLE

Example 1.1: Program to print Hello World

Use the Code 1.1 to print strings using Python:

```
print("Hello World")

print("Python is opensource")
```

The output of the Example 1.1 is shown in Figure

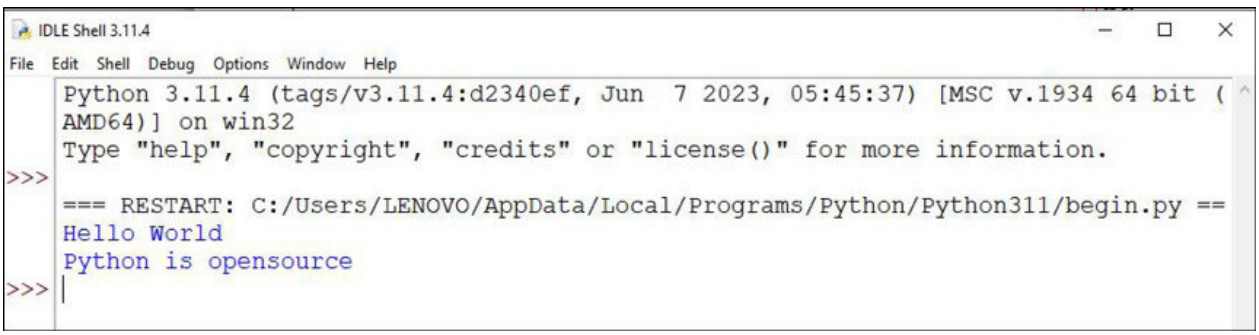

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
=== RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/begin.py ==
Hello World
Python is opensource
>>>

Figure 1.12: Output of Hello World Example 1.1

Variable declaration and getting user input in Python

Python has no command for declaring a variable. Variables are the memory locations, and variables do not need to be declared with any particular type and can even change type after they have been set.

Note: The variables are case-sensitive and are used to store data values. Variable names consist of letters like a-z, A-Z, and numbers (0-9), or variable names may start with a letter or underscore(_).

Multiple variables can be assigned in a single line of code using Python. For example: a,b,c = 3,5,6

Example 1.2: Example of user input

In this example, the name is a variable, and input is a keyword to be taken as input from a user. Execute the following code in Code

name= input("Enter your name")

print("Hello" + format(name))

time=input("Enter the appointment time")

print(time)

Note: In Example 1.2, input is used as the keyword. The format () function used in the program is a built-in function that returns the formatted representation of the given value; that is, it returns a formatted string.

Example 1.2 in the IDLE shell is shown in Figure

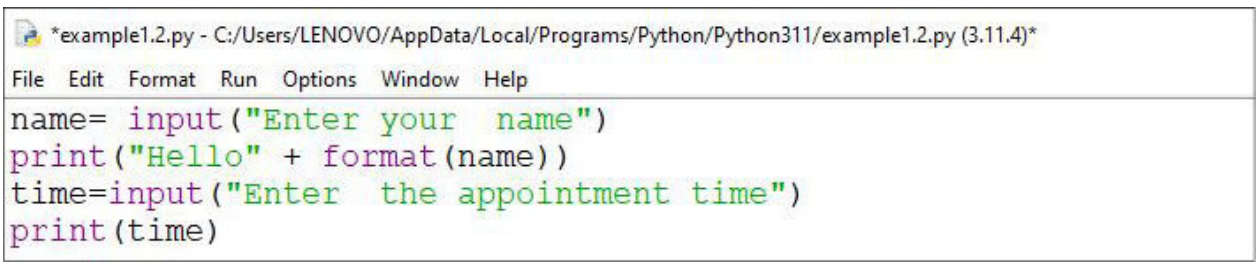


Figure 1.13: User input from the command line

The output of the Example 1.2 is shown in Figure

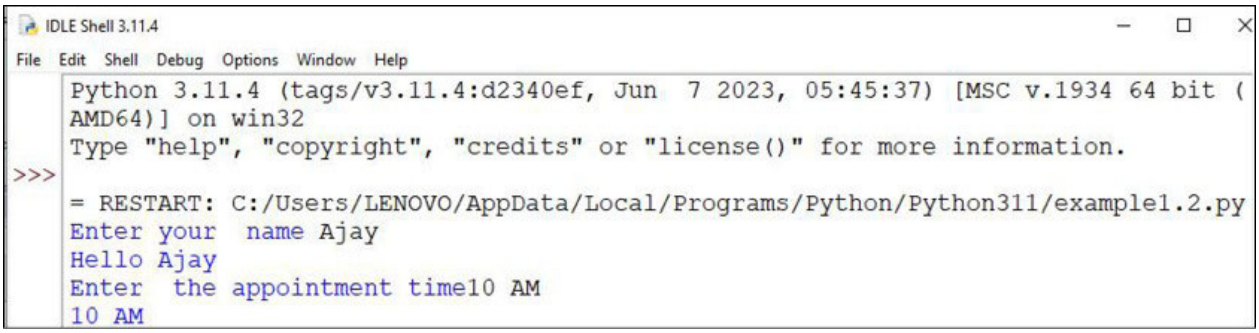

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/example1.2.py
Enter your name Ajay
Hello Ajay
Enter the appointment time10 AM
10 AM

Figure 1.14: User input from the command line

Example 1.3: Addition of two numbers

In Example two numbers are added using variables, as shown in Figure The variables a and b are converted into integers to get the sum.

Code 1.3:

```
a = input("Enter  the first number")

print ("The value of a is", a  )

b = input("Enter  the second number")

print("The value of b is", b)

c = int(a) + int(b)

print ("The sum is:" , c)
```

The IDLE shell is shown in the following figure:

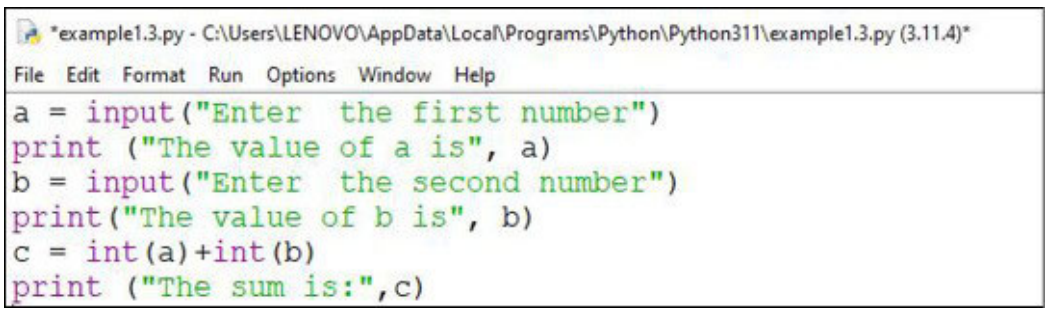

example1.3.py - C:\Users\LENOVO\AppData\Local\Programs\Python\Python311\example1.3.py (3.11.4)
File Edit Format Run Options Window Help
a = input("Enter the first number")
print ("The value of a is", a)
b = input("Enter the second number")
print("The value of b is", b)
c = int(a)+int(b)
print ("The sum is:",c)

Figure 1.15: Addition of two numbers

The output of the Example 1.3 is shown in Figure

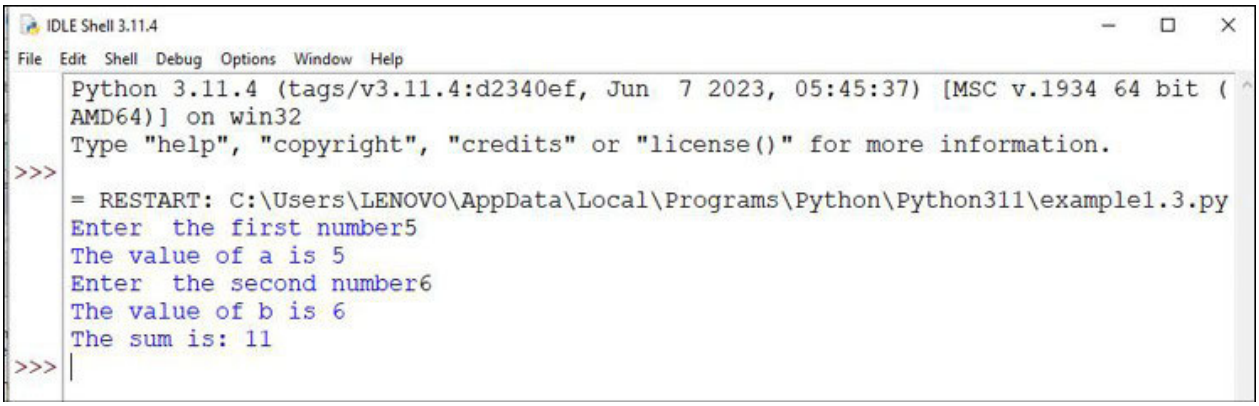

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\LENOVO\AppData\Local\Programs\Python\Python311\example1.3.py
Enter the first number5
The value of a is 5
Enter the second number6
The value of b is 6
The sum is: 11
>>>

Figure 1.16: Addition of two numbers

Example 1.4: Write a program to swap two values

In this example, values of the x and y variables are swapped using a temporary variable, as shown in Figure

Code 1.4:

```
x=5

y=10

temp=x

x=y

y=temp

print ('The value of x after swapping:{}'.format(x))

print('The value of y after swapping:{}'.format(y))
```

Note: The format () method is used for String formatting, and the {} symbol is a placeholder to store specific variables and expressions within it. In Figure the x variable value is displayed, where {} placeholder is used.

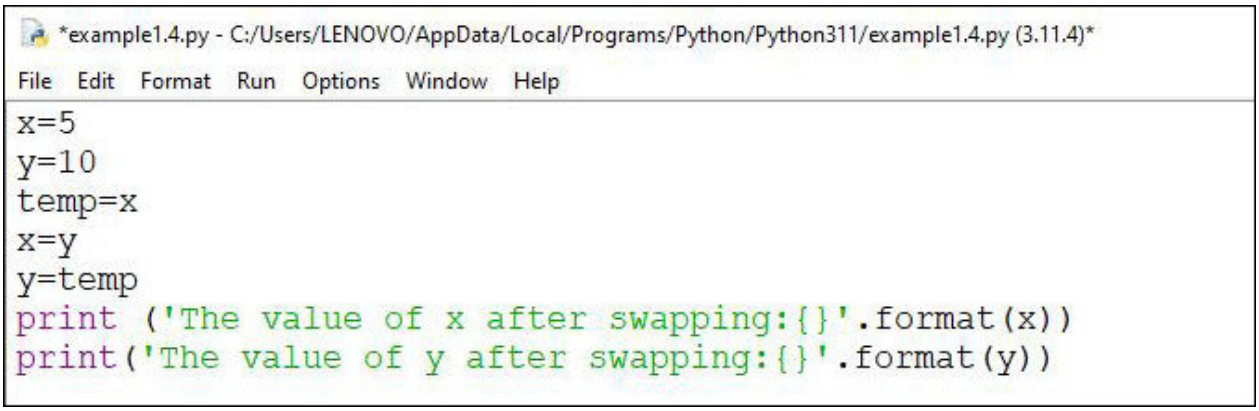


Figure 1.17: Swapping the variable values

The output of the preceding program is shown in Figure

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for m
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python
The value of x after swapping:10
The value of y after swapping:5
>>>

Figure 1.18: Swapping of two values

Python datatypes

Unlike other programming languages, Python does not need the declaration of datatypes. Python comes with several built-in datatypes that let you store and work with various types of data. Among Python's datatypes are:

Numeric types, such as and integer are included.

Types for sequences include and set types.

Binary types consist of bytes and byte arrays.

Different date and time formats, such as datetime, date, time, and so on.

Example 1.5: Example of integer, float, string datatypes, and type conversion

In this example, name and height are the variables and are of integer, float and string datatype. The integer float variables are converted to string format using str keyword, that is called type conversion.

Code 1.5:

```
age = 25  # Integer variable

name = "ABC"  # String variable

height = 5.9  # Float variable

# Type Conversion

# Integer to String

str_age = str(age)

print("Age in string form is:", str_age)

# Float to String

str_height = str(height)
```

print("Height in string form is:", str_height)

The program window is shown in Figure

```
age = 25  # Integer variable
name = "ABC"  # String variable
height = 5.9  # Float variable

# Type Conversion
# Integer to String
str_age = str(age)
print("Age in string form is:", str_age)
# Float to String
str_height = str(height)
print("Height in string form is:", str_height)
```

Figure 1.19: Program window of datatypes and type conversion

The output of Example 1.5 is shown in Figure

```
>>>
========== RESTART: E:\python revised\pythonprograms\a
Age in string form is: 25
Height in string form is: 5.9
>>>
```

Figure 1.20: Output window of datatype and type conversion

Operators in Python

The operators in Python are as follows:

Arithmetic operators

Relational operators

Logical, Bitwise, and assignment operators

Identity operators

Membership operators

Example 1.6: Arithmetic operators in Python

In this example, the arithmetic operators are explained using the Code 1.6:

```
# Basic operators in Python

a = int(input("Enter the first number"))

b = int(input("Enter the second number"))

addition = a+b    #  addition

sub = a–b    # subtraction

multiplication = a*b    # multiplication

division = a/b      # division

remainder = a%b    # modulus

print("Addition" , addition)

print("Subtraction", sub)

print("Multiplication is", multiplication)
```

print("Division" , division)

print("Remainder", remainder)

The program window for the Code 1.6 is as follows:

```
# Basic operators in Python

a = int(input("Enter  the first number"))
b = int(input("Enter  the second number"))

addition = a + b                #addition
sub = a-b                       #subtraction
multiplication = a * b          #multiplication
division = a/b                  #division
remainder = a%b                 #modulus

print("Addition" , addition)
print("Subtraction", sub)
print("Multiplication is", multiplication)
print("Division" , division)
print("Remainder", remainder)
```

Figure 1.21: Arithmetic operators in Python

The output of Example 1.6 is shown in Figure

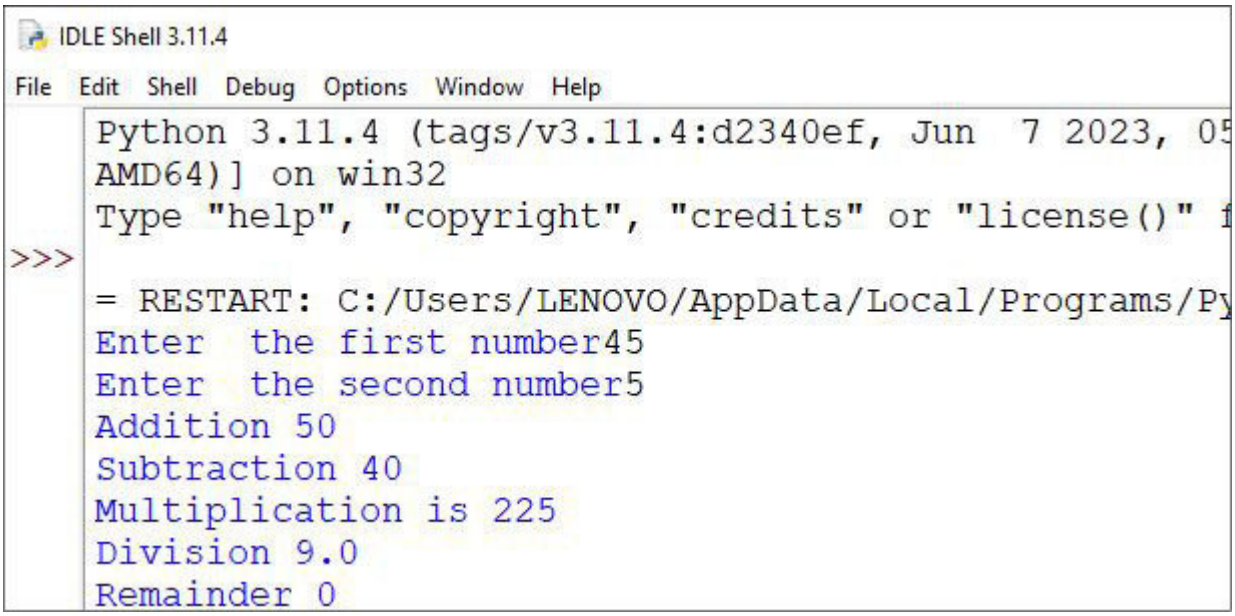


Figure 1.22: Output of arithmetic operators in Python

Note: # is used to comment the statement.

Relational operators

The relational operators compare the values of two operands. The relational operators shown in the example are:

Greater than >

== equal to

!= not equal to

>= Greater than or equal to

<= less than or equal to

Example 1.7: Relational operators in Python

In this example, relational operators like greater than, less than, equal to, or not equal to are performed.

Code 1.7:

```
# Relational operators in Python

a = input("Enter  the first number")

print ("The value of a is" +format(a))

b = input("Enter  the second number")

value of b is" + format(b))

if a > b:

  print ('a is greater than b',a>b)

else:

  print ('a is not greater than b',a>b)

if a==b:
```

```
  print('a is equal to b',a==b)
else:
  print('a is not equal to b',a==b)
if a!=b:
  print('a is not equal to b',a!=b)


if a>=b:
  print('a is greater than equal to b',a>=b)
else:
  print('a is not greater than equal to b',a>=b)
if a<=b:
  print('a is less than equal to b',a<=b)
else:
  print('a is not less than equal to b',a<=b)
```

The program window is shown in Figure

```
# Relational operators in Python
a = input("Enter  the first number")
print ("The value of a is" +format(a))
b = input("Enter  the second number")
print("The value of b is" + format(b))
if a > b:
   print ('a is greater than b',a>b)
else:
   print ('a is not greater than b',a>b)
if a==b:
   print('a is equal to b',a==b)
else:
   print('a is not equal to b',a==b)
if a!=b:
   print('a is not equal to b',a!=b)
if a>=b:
   print('a is greater than equal to b',a>=b)
else:
   print('a is not greater than equal to b',a>=b)
if a<=b:
   print('a is less than equal to b',a<=b)
else:
   print('a is not less than equal to b',a<=b)
```

Figure 1.23: Example of the relational operator

The output of Example 1.7 is shown in Figure

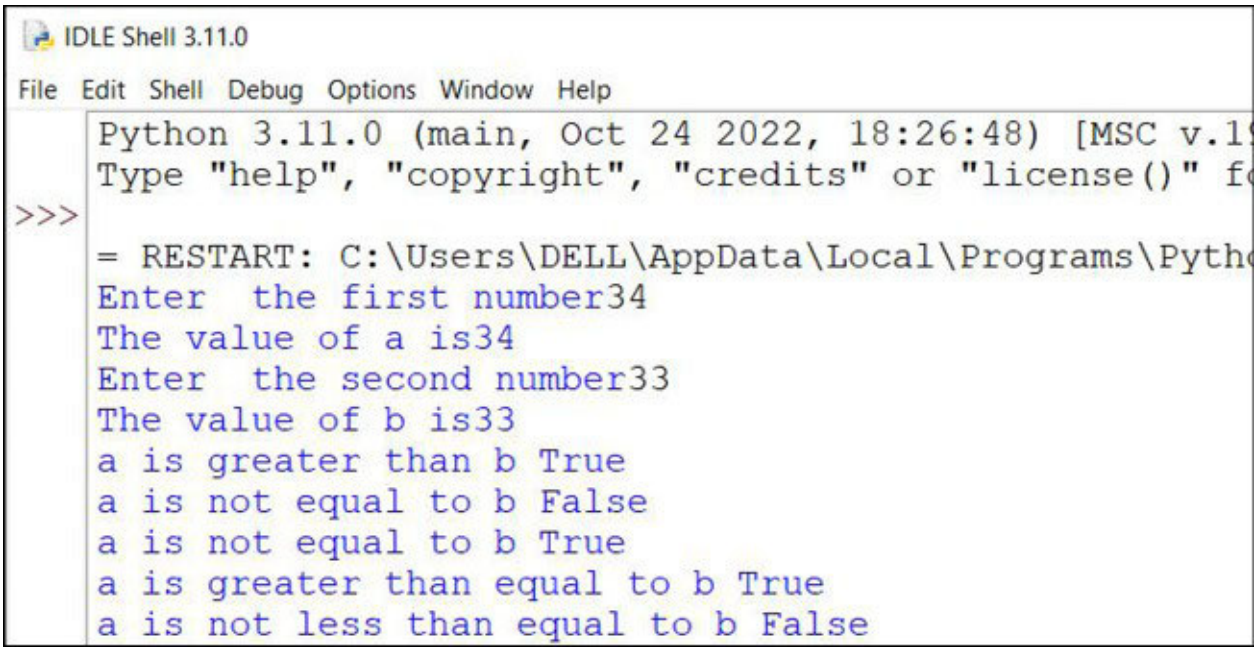

IDLE Shell 3.11.0
File Edit Shell Debug Options Window Help
Python 3.11.0 (main, Oct 24 2022, 18:26:48) [MSC v.1
Type "help", "copyright", "credits" or "license()" f
>>>
= RESTART: C:\Users\DELL\AppData\Local\Programs\Pyth
Enter the first number34
The value of a is34
Enter the second number33
The value of b is33
a is greater than b True
a is not equal to b False
a is not equal to b True
a is greater than equal to b True
a is not less than equal to b False

Figure 1.24: Output of relational operators

Logical, bitwise and assignment operators

The logical operators in Python are and Bitwise operators include Bitwise AND Bitwise OR and Bitwise XOR whereas assignment operators include assignment addition assignment division assignment subtraction assignment and multiplication assignment

Example 1.8 Example of logical, bitwise, left shift, right shift and assignment operators

In this example, logical operators that are not are performed. Along with logical operators, bitwise operators, and assignment operators are performed.

Code 1.8:

```
# Logical and Bitwise operators in Python

a = 1

b = 0

# "and" operator will check both the conditions i.e.if one condition is true and

# another is  false,then output will be false.

print (a and b)  # output will be printed 0 i.e.false

# In "or" operator, one condition is true and another is false,then output will be true.

print (a or b)    # output will be printed 1 i.e.true

print (not a)    # output will be "False"
```

```
c = int(input("Enter  the value"))

print ("The value of c is" , c)

d = int(input("Enter  the value"))

print("The value of d is" ,d)



# Bitwise And(&) operator

print('Result after Bitwise AND operation is',c&d)

# Bitwise or(|) operator

print('Result after Bitwise OR operation is',c|d)

#Bitwise XOR operator

print('Result after Bitwise XOR operation is',c^d)

# Bitwise Right Shift operator

print('Result after Right Shift operation is',c>>1)

# Bitwise Left Shift Operator

print('Result after Left Shift operation is',c<<1)
```

Assignment operator

c+=10

print('c value after adding 10 using Assignment operator is',c)

c-=3

print('c value after subtracting 3 using Assignment operator is',c)

The program window is shown in Figure

```
#Logical and Bitwise operators in Python
a=1
b=0
#"and" operator will check both the conditions i.e.if one condition is true and
print(a and b)#output will be printed 0 i.e.false
#In "or" operator,one condition is true and another is false,then output will be
print(a or b) #output will be printed 1 i.e.true
print(not a)  #output will be "False"
c=int(input("Enter the value"))
print("The value of c is",c)
d=int(input("Enter the value"))
print("The value of d is",d)
#Bitwise And(&)operator
print('Result after Bitwise AND operation is',c&d)
#Bitwise or(|) operator
print('Result after Bitwise OR operation is',c|d)
#Bitwise XOR operator
print('Result after Bitwise XOR operation is',c^d)
#Bitwise Right Shift operator
print('Result after Right Shift operation is',c>>1)
#Bitwise Left Shift Operator
print('Result after Left Shift operation is',c<<1)
#Assignment operator
c+=10
print('c value after adding 10 using Assignment operator is',c)
c-=3
print('c value after subtracting 3 using Assignment operator is',c)
```

Figure 1.25: Example of logical, bitwise, left shift, right shift and assignment operators

The output of Example 1.8 is shown in Figure

```
= RESTART: C:\Users\DELL\AppData\Local\Programs\Python\Python311\example1.7.py =
0
1
False
Enter the value5
The value of c is 5
Enter the value4
The value of d is 4
Result after Bitwise AND operation is 4
Result after Bitwise OR operation is 5
Result after Bitwise XOR operation is 1
Result after Right Shift operation is 2
Result after Left Shift operation is 10
c value after adding 10 using Assignment operator is 15
c value after subtracting 3 using Assignment operator is 12
```

Figure 1.26: Output of logical, bitwise, left shift, right shift and assignment operators

In and operator will check both the conditions; that is, if one condition is true and another is false, then the output will be false.

In the or operator, if one condition is true and another is false, the output will be true.

The bitwise and operation output value is 4; the explanation of output is mentioned below:

c = 5 (Binary value of 5 is: 101)

d = 4 (Binary value of 4 is: 100)

output = 100 (Decimal value of binary value 100 is 4),

Since 1 and 1 becomes 1 (both are true), 0 and 0 results in 0, 1 and 0 results in 0, because in and operation, if one value is true and another is false, then result is false.

The bitwise or operation output value is 5. The explanation of the output is mentioned below:

c = 5(Binary value of 5 is:101)

d = 4(Binary value of 4 is:100)

output = 101(Decimal value of binary value 101 is 5),

Since 1 or 1 becomes 1 (both are true), 0 or 0 results in 0, 1 or 0 results in 1, and in or operation, either of the conditions should be true; if one value is true and another is false, even then the result is true.

The bitwise xor operation output value is 1, the explanation of the output is mentioned below:

c = 5(Binary value of 5 is:101)

d = 4(Binary value of 4 is:100)

output = 001(Decimal value of binary value 001 is

Since 1 xor 1 becomes 1 (both values are the same. Therefore output is 0), 0 xor 0 results in 0, and 1 xor 0 results in 1 (output is 1 because there is a

change in values; one value is 1 and other value is 0, therefore, output as per xor is 1).

In the bitwise right shift operator, the value is shifted by 1 towards the right side. The right shift operation of c>>1 variable value: 5>>1 output is 2, because binary value of 5 is: 101, on right shifting the digits by 1 result into binary value ‘10’ which is the binary value of 2.

In Bitwise Left Shift the value is shifted by 1 towards the left side. The left shift operation of c<<1 variable value: 5<<1 output is 10, because binary value of 5 is:101, on left shifting the digits by 1, it becomes 1010, which is the binary value of 10.

Identity operators

In Python, there are identity operators as follows:

a boolean value true if the values in variables are the same; else a false value will be returned.

a boolean value true if the values in the variables are not the same.

Example 1.9: Example of is and isnot identity operator

In this example, if the values in variables c and d are the same, then the is identity operator returns the true value.

Variable a and b contain the same value; therefore, the isnot operator will return the output as true because the values are not the same. Example 1.9 is shown in Figure

Code 1.9:

```
c = input("Enter  the value")

print ("The value of c is" +format(c))

d = input("Enter  the value")

print("The value of d is" + format(d))

print('Output is true because c and d values are identical',c is d)

a = 'java'

b = 'php'

print ('Output is true because a and b values not matches',a is not b)
```

a1 = 'java'

b1 = 'java'

print ('Output is false because a1 and b1 values matches',a1 is not b1)

```
File  Edit  Format  Run  Options  Window  Help
c = input("Enter  the value")
print ("The value of c is" +format(c))
d = input("Enter  the value")
print("The value of d is" + format(d))
print ('Output is true because c and d values are identical',c is d)
a = 'java'
b = 'php'
print ('Output is true because a and b values not matches',a is not b)
a1 = 'java'
b1 = 'java'
print ('Output is false because a1 and b1 values matches',a1 is not b1)
```

Figure 1.27: Program window of identity operator

The output of Example 1.9 is shown in Figure

```
>>>
= RESTART: C:/Users/DELL/AppData/Local/Programs/Python/Python311/example1.8.py =
Enter  the value5
The value of c is5
Enter  the value5
The value of d is5
Output is true because c and d values are identical True
Output is true because a and b values not matches True
Output is false because a1 and b1 values matches False
```

Figure 1.28: Output of identity operators

Membership operators

There are two membership operators in Python, which are as follows:

in - Output will be true if the variable value is found in the specified sequence. Else, the output will be false.

not in - Output will be true if the variable value is not in the specified sequence. Else, the output will be false.

Note: These operators are used in strings, lists, and tuples.

Example 1.10: Using the membership operator

The creation of a list including elements, for instance, is shown in Figure The membership operators in and not in will yield boolean values indicating whether or not the components are included in the

Code 1.10:

```
c = int (input("Enter  the value"))

print ("The value of c is",c)

d = int(input("Enter  the value"))

print("The value of d is", d)

#  A list is created and the elements are placed into it.

lst1 = [12,3,5,6,9]

If c in lst1:

  print('c value is available in the list',c in lst1)

else:
```

print('c value is not available in the list',c in lst1)

if d not in lst1:

print('d value is not available in the list',d not in lst1)

else:

print('d value is available in the list',d not in lst1)

The program window for the same is shown below:

```
c = int(input("Enter  the value"))
print ("The value of c is",c)
d = int(input("Enter  the value"))
print("The value of d is",d)

lst1=[12,3,5,6,9] #A list is created and the elements are placed into it.
if c in lst1:
   print('c value is available in the list',c in lst1)
else:
   print('c value is not available in the list',c in lst1)
if d not in lst1:
  print('d value is not available in the list',d not in lst1)
else:
  print('d value is available in the list',d not in lst1)
```

Figure 1.29: Program window of Membership operator

The output of Example 1.10 is shown in Figure

```
>>>
    = RESTART: C:/Users/DELL/AppData/Local/Programs/Python/Python311/example1.9.py =
    Enter  the value5
    The value of c is 5
    Enter  the value10
    The value of d is 10
    c value is available in the list True
    d value is not available in the list True
```

Figure 1.30: Output of membership operator

Precedence of operators

The order in which operators are evaluated in an expression in Python is determined by operator precedence. Higher precedence operators are assessed first. When operators are equal in priority, the evaluation sequence is typically left to right. For example, in a statement var=1+2*3, Multiplication (*) has higher precedence, therefore 2*3 will be evaluated first. Here is a list of operators in Python, ordered by precedence from highest to lowest as shown in Table

Table 1.1: Precedence of operators from highest to lowest

type() function in Python

type() function is used in Python for determining the datatype or class of a variable. Using this function helps us to find the datatype of a variable because, in python, the variable datatype may not to be given explicitly.

Example 1.11: Write a program for the type() function

In this example, three variables are declared, and for finding the type of these variables, type() function is used.

Code 1.11:

```
var1 = 15

var2 = "Welcome in Python"

var3 = [12, 23, 43]

print("It is of type Integer",type(var1))

print("It is of type String",type(var2))

print("It is of type List",type(var3))
```

The program window is shown in Figure

```
var1 = 15
var2 = "Welcome in Python"
var3 = [12, 23, 43]

print("It is of type Integer",type(var1))
print("It is of type String",type(var2))
print("It is of type List",type(var3))
```

Figure 1.31: Example of type () function in Python

The output of the preceding example is shown in Figure

```
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/
y
It is of type Integer <class 'int'>
It is of type String <class 'str'>
It is of type List <class 'list'>
```

Figure 1.32: Output of type() function in Python

Conclusion

This chapter covered an introduction to Python programming language and its installation in an easy-to-understand way with a beginner's mindset. Variable declaration, data types, and their conversion are all discussed. Using appropriate examples, the Python operators are explored and explained. In the next chapter, we will cover the basics of control statements, loops, the range() function, and other programming concepts.

Points to remember

Variables are used to store data values. Variables are the names that point to objects in memory.

Python is case-sensitive.

Multiple variables can be assigned in a single line of code using Python.

In Python, like other languages, we need not to declare a variables data type separately. Therefore, Python language is dynamically typed. For Example: If x = it means, an integer value is assigned to x and if y = it means string value is assigned to y.

input() function is used in Python to take an input from a keyboard.

format() method is used for string formatting and the variable values and expressions are stored inside {} expression.

type() function is used in python for determining the datatype or class of a variable.

Exercises

Is Python a compiler-based or an interpreter-based language? What do you mean by a variable?

Why is Python dynamically typed? Explain.

Discuss relational and logical operators in Python with the help of suitable examples.

What will be the output of the following statement?

```
a = 5

b = 45

print (5+45)
```

What will be the output of the following statement?

```
a = 5

b = 45

print ("5+45")
```

What will be the output of the following statement?

```
a = int(input("Enter the value of variable a: "))

b = int(input("Enter the value of variable b: "))

a, b = b, a

print(f"After swapping, a = {a} and b = {b}")
```

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 2

Conditions and Loops

Introduction

In Python and several other programming languages, conditions and loops are fundamental programming elements. They enable you to make decisions, repeat activities, and manage the flow of your program. For conditions, we use if, elif, and else structures. We will learn about the for and while loops utilizing the range() function. Loops are crucial for managing the flow of programs and effectively carrying out repetitious operations. In addition to that, control statements like break and continue are covered with the aid of relevant examples.

Structure

In this chapter, the following topics will be covered:

If condition

Nested-If condition

About loops

Range() function in loops

Break and continue statements

Objectives

At the end of this chapter, you will be able to understand the fundamentals of control statements and implement the logic using if conditions and loops wherever it is necessary. Apart from that, you will be able to learn the difference between break and continue statements.

If condition

In Python, the elif keyword is used instead of else if for multiple if conditions. The elif keyword in Python is interpreted as follows: If the previous conditions are not true, then try another

The if_ _ elif _ _ else statement is used to execute a statement or a block of statements conditionally.

Example 2.1: Example of if condition

In this example, a basic implementation of the if condition is shown to learners. A value input as an integer will be passed in a variable If a value passed inside a num variable is positive, then the first condition will be checked, and the output will be displayed: The number contains the positive If a value passed inside a num variable is negative, then the elif condition will execute, and the output will be displayed as The number contains the negative If the variable num contains neither the positive nor negative value, then else condition will execute and output will be displayed as The number contains zero

Code 2.1:

```
num = int(input("Enter a number: "))

if num > 0:

print("The number contains the positive value.")

elif num < 0:

print("The number contains the negative value.")

else:

print("The number contains the zero value.")
```

The program window of Example 2.1 in the IDLE shell is shown in Figure

```
num = int(input("Enter a number: "))
if num > 0:
 print("The number contains the positive value.")
elif num < 0:
 print("The number contains the negative value.")
else:
 print("The number contains the zero value.")
```

Figure 2.1: Program window of if condition

The output of Example 2.1 is shown in Figure

```
Enter a number: 3
The number contains the positive value.
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/
Enter a number: -1
The number contains the negative value.
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/
Enter a number: 0
The number contains the zero value.
```

Figure 2.2: Output window of if…elif condition

Example 2.2: Example of if condition using and, or operators

In this example, and c are the variables and contain the boolean value The first if condition will print(1) if variable a value is False and variable b is True. However, this condition is false because, in the and operator, both conditions should be true to get the output.

In the second if condition, all the conditions are true, therefore, the output will be ‘2’.

The third condition is again false because, in the or operator, either of the conditions should be true; that is, if one condition is true and the other is false, the output will be

```
a = True

b = True

c = True

if not a and b:

    print 1

elif  a and b and c:

    print 2
```

elif not a or b or not b and a:

    print 3

else:

    print 4

The program window of Example 2.2 in the IDLE shell is shown in Figure

```
a = True
b = True
c = True
if not a and b:
    print(1)
elif  a and b and c:
    print(2)
elif not a or b or not b and a:
    print(3)
else:
    print(4)
```

Figure 2.3: Program window of If condition using ‘and’, ‘or’ operators

The output window of Example 2.2 is shown in Figure

```
>>>
= RESTART: C:/Users/DELL/AppData/Local/Programs/Pythc
2
>>>
```

Figure 2.4: Output window of if condition using ‘and’, ‘or’ operators.

Note: The statement “if not a and b:” means if neither a is True and b is True. In ‘and’, all the conditions should be true; only then output is true, whereas in ‘or’, at least one condition should be true.

Example 2.3: Example of checking conditions using and, or operators

In this example, in the first if condition, both the conditions are true; therefore, as per the and operator, if both conditions are true, then the output is true. In the second if condition, either of the conditions is true. Therefore, as per the or operator, if either of the condition is true, then the output is true.

Code 2.3:

```
name="vijay"

age=23

if name=="vijay" and age==23:

print("your name is vijay and you are  23 years old")

if name=="vijay" or name=="sohan":

    print("your name is either vijay or sohan")
```

The program window is shown in Figure

```
name="vijay"
age=23
if name=="vijay" and age==23:
    print("your name is vijay and you are  23 years old")
if name=="vijay" or name=="sohan":
    print("your name is either vijay or sohan")
```

Figure 2.5: Program window of checking conditions using and, or operators

The output of Example 2.3 is shown in Figure

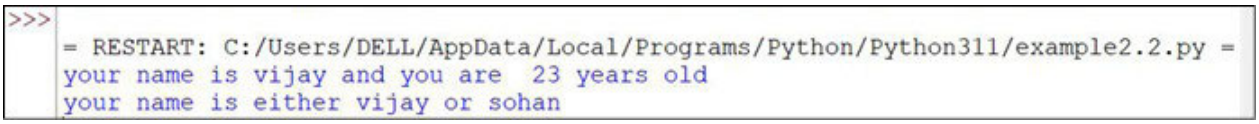

>>>
= RESTART: C:/Users/DELL/AppData/Local/Programs/Python/Python311/example2.2.py =
your name is vijay and you are 23 years old
your name is either vijay or sohan

Figure 2.6: Output window of checking conditions using ‘and’, ’or’ operators

Nested-if condition

In programming, the term nested if condition describes a scenario in which one or more if statements are nested inside another if statement. This enables us to write code with complex decision-making structures. You can build a hierarchy of conditions using nested if statements, where the execution of some code is reliant upon the fulfillment of several requirements.

Example 2.4: Example of nested-if conditions to find the greatest of three numbers

The outer if statement in this example determines whether num1 is greater than In the event that it is true, num1 and num3 are compared in the inner if statement. The inner else sentence compares num2 with num3 if the outer if statement is false, which indicates that num2 is greater than or equal to It prints the highest number out of the three, depending on the conditions.

Code 2.4:

```
num1 = int(input("Enter first number: "))

num2 = int(input("Enter second number: "))

num3 = int(input("Enter third number: "))

if num1>num2:

    if num1>num3:

print("The greatest number is num1:",num1)

    else:

        print("The greatest number is num3:",num3)

else:
```

if num2>num3:

print("The greatest number is num2:",num2)

else:

print("The greatest number is num3:",num3)

The program window of Example 2.4 is shown in Figure

```
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
num3 = int(input("Enter third number: "))

if num1>num2:
    if num1>num3:
        print("The greatest number is num1:",num1)
    else:
        print("The greatest number is num3:",num3)
else:
    if num2>num3:
        print("The greatest number is num2:",num2)
    else:
        print("The greatest number is num3:",num3)
```

Figure 2.7: Program window of greatest of three numbers using nested-if condition

The program is checked for all the three numbers as shown in output in Figure

```
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs
Enter first number: 34
Enter second number: 45
Enter third number: 56
The greatest number is num3: 56
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs
Enter first number: 56
Enter second number: 34
Enter third number: 3
The greatest number is num1: 56
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs
Enter first number: 67
Enter second number: 89
Enter third number: 3
The greatest number is num2: 89
```

Figure 2.8: Output window of greatest of three numbers using nested-if condition

Loops

Loops are instructions that repeat until the given condition is satisfied or true. The advantage of loops is that they execute multiple statements and avoid the unnecessary repetition of the block of statements. The examples of while and for loop are discussed below.

Example 2.5: Example of a while loop

In the while loop, the condition is checked first at the entry point, and if the condition is true, then the statements inside the loop will execute.

Code 2.5:

```
i = 0

while i < 3:

    print i

    i=i + 1
```

The program window is shown in Figure

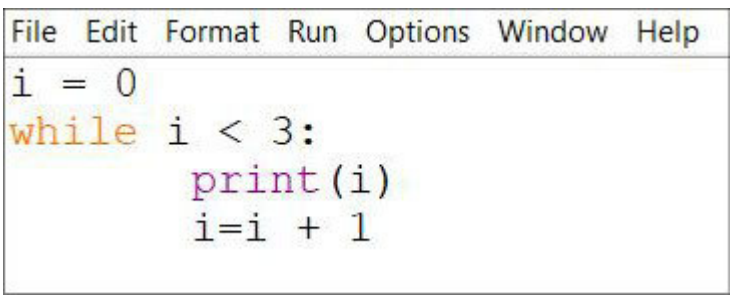


Figure 2.9: While loop

The output of Example 2.5 is shown in Figure

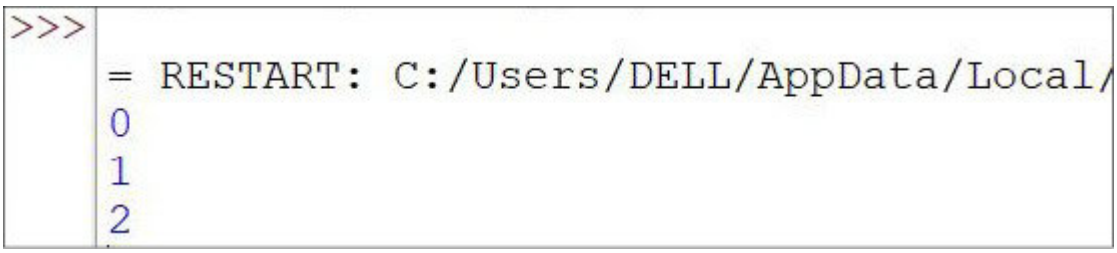


Figure 2.10: Output of a while loop

Example 2.6: Sum of first five numbers using the while loop

In this example, a variable counter is initialized to 1, and the while loop will iterate from 1 to 5. The sum of the first five numbers will be displayed as an output. The variables declared are n, s, and counter. The while loop will check the condition first, and if the condition is true, then statements inside the while loop will get executed. The loop will run until the counter value is less than or equal to 5.

Code 2.6:

```
n=5

s=0

counter=1

while counter<=n :

    s=s+counter

    counter=counter+1

print("Sum of 5 numbers is:%d" %(s))
```

The program window of Example 2.6 is shown in Figure

```
n=5
s=0
counter=1
while counter<=n :
    s=s+counter
    counter=counter+1
print("Sum of 5 numbers is:%d" %(s))
```

Figure 2.11: Program window of finding the sum of first five numbers using the while loop

The output of the Example 2.6 is shown in Figure

```
>>>
= RESTART: C:/Users/DELL/AppData/Local/Program
Sum of 5 numbers is:15
>>>
```

Figure 2.12: Output window of finding the sum of first five numbers using the While loop

Example 2.7: Demonstrating break statement in a while loop

In this program, the count variable is initialized to zero. Here, the while loop will run until the condition is true. The break statement will terminate the loop when the condition becomes greater than or equal to five. Here, on using the break statement, the loop will terminate, and further iterations will be stopped.

Code 2.7:

```
count = 0

while True:

    print(count)

    count += 1

# count = count + 1  i.e. Assignment operator

    if count >= 5:

        break
```

The program window of Example 2.7 is shown in Figure

```
count = 0
while True:
    print(count)
    count += 1       # count=count+1 i.e.Assignment operator.
    if count >= 5:
        break
```

Figure 2.13: Program window of while loop and break statement

The output of Example 2.7 is shown in Figure

```
>>>
= RESTART: C:/Users/DELL/AppData/Local/Programs/Python/Python311/exa
0
1
2
3
4
```

Figure 2.14: Output window of while loop and break statement

Example 2.8: Example for finding a palindrome number using a while loop

In this example, the reverse of a number is performed using a while loop. The loop will iterate until the number entered becomes 0, and the remainder of a number is obtained using a modulus operator. If the reverse of a number is the same as that of the number entered, then the number is a palindrome number else, not a palindrome number.

Code 2.8:

```
n = int(input("Enter the Number"))

i = int(n)

rev = 0

# while loop will execute only if the condition is true

while(i > 0):

#  remainder is obtained using % (modulus) operator

rem = i % 10

  rev = rev*10 + rem

  i = i//10
```

print("The reverse of a no.is",rev)

if(n==rev):

print(str(n) + "is a palindrome number")

else:

print(str(n) + "isn't a palindrome number")

The program window of Example 2.8 is shown in Figure

```
n = int(input("Enter the Number"))
i = int(n)
rev = 0

while(i>0):  #while loop will execute only if the condition is true
   rem=i%10  #remainder is obtained using % (modulus) operator
   rev=rev*10+rem
   i = i//10


print("The reverse of a no.is",rev)

if(n==rev):
 print(str(n) + "is a palindrome number")
else:
 print(str(n) + "isn't a palindrome number")
```

Figure 2.15: Program window of finding a palindrome number using while loop

The output Window of Example 2.8 is shown in Figure

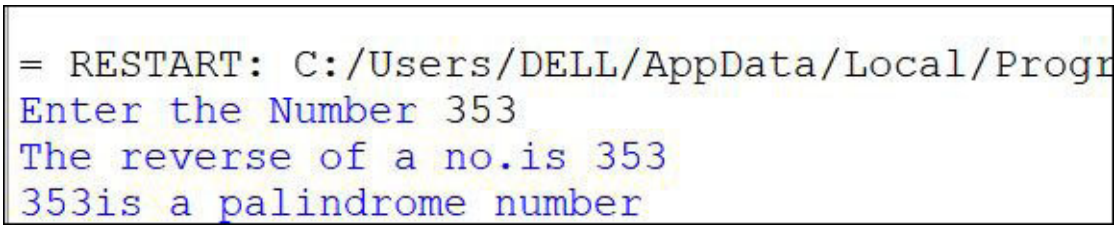


Figure 2.16: Output window of finding a palindrome number using while loop

Example 2.9: Multiplication of a number using a while loop

In this example, variables and i are declared. Here, while loop will iterate till i is not equal to 0. The result variable is updated every time by adding a 5 value to it and printing the value of every step, and i value is decremented. It will exit the loop when i value becomes equal to 0, and when we get out of the loop, multiplication is performed to get the same output of a while loop i.e. 5*5=25.

Code 2.9:

```
#variable is declared

n = 5

#variable is initialized to 0

result = 0

# n value is passed to i

i = n

#loop will iterate until i is not equal to 0

while(i!=0):
```

result=result+n

print(str(i) + ' ' + 'loop value is' + ' ' + str(result))

i value will decrement one by one

i=i-1

print(str(n) + '*' + str(n) + '=' + str(result))

The program window of Example 2.9 is shown in Figure

```
n = 5  #variable is declared
result = 0 #variable is initialized to 0
i = n  # n value is passed to i
while(i!=0): #loop will iterate until i is not equal to 0
    result=result+n
    print(str(i) + ' ' + 'loop value is' + ' ' + str(result))
    i=i-1 # i value will decrement one by one

print(str(n) + '*' + str(n) + '=' + str(result))
```

Figure 2.17: Program window of multiplication of a number using a while loop

The output window of Example 2.9 is shown in Figure

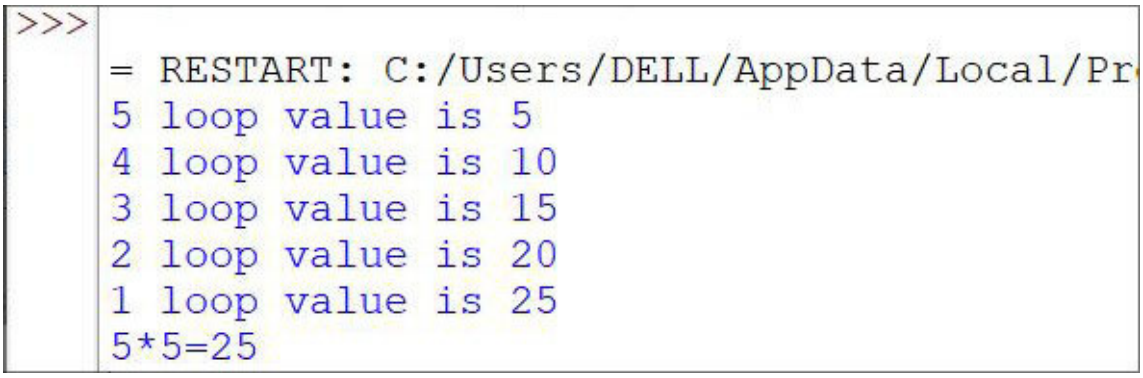


Figure 2.18: Output window of multiplication of a number using a while loop

Example 2.10: Example of for loop

In this program, a list is declared, and the for loop will iterate over a sequence of elements in a list and move from the index.

Code 2.10:

```
# A list is declared

languages = ["C","C++","Perl","Python"]

# for loop iterates over a sequence of elements in a list

# and there is no need to set the index

for x in languages:

    print(x)
```

The program window is shown in Figure

```
languages = ["C","C++","Perl","Python"]      # A list is declared
for x in languages: # for loop iterates over a sequence of elements in a list
    print(x)
```

Figure 2.19: Program window of a for loop

The output of Example 2.10 is shown in Figure

```
>>>
= RESTART: C:/Users/DELL/AppData/Loca
C
C++
Perl
Python
```

Figure 2.20: Output window of a for loop

Example 2.11: Example of the list and for loop

In this example, a list is declared, and on the index, java is there, and on the index, the dotnet value is there. For loop will iterate through a loop, and if values java and dotnet exist in a list, the output will be printed.

Code 2.11:

```
course = ["java","dotnet"]

for a in course:

    if a == "java":

        print("please do java")

    else:

        print("prefer to do dotnet also")
```

The program window is shown in Figure

```
File  Edit  Format  Run  Options  Window  Help
course = ["java","dotnet"]
for a in course:
    if a == "java":
        print("please do java")
    else:
        print("prefer to do dotnet also")
```

Figure 2.21: Program window of implementing lists using for loop

The output of Example 2.11 is shown in Figure

```
>>>
= RESTART: C:/Users/DELL/AppData/Local/Proc
please do java
prefer to do dotnet also
```

Figure 2.22: Output window of implementing lists using for loop

Example 2.12: Print the numbers using range() function in for loop

In this program, range() function returns the sequence of numbers. The range starts from 0 by default, and value will be incremented by 1 by default.

Code 2.12:

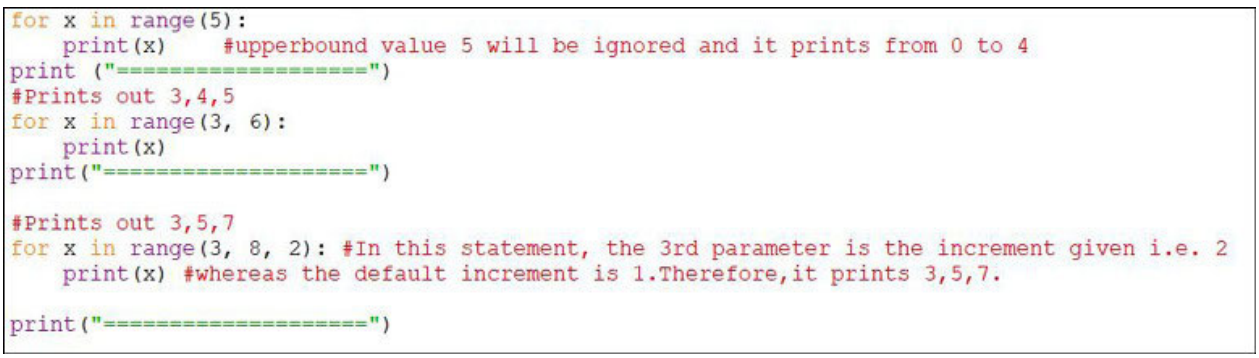
```
for x in range(5):

# upperbound value 5 will be ignored and it prints from 0 to 4

    print(x)

print("====================")

# Prints out 3,4,5

for x in range(3, 6):

    print(x)

print("====================")

# In this statement, the 3rd parameter is the increment given i.e. 2

# whereas the default increment is 1. Therefore, it prints 3,5,7
```

for x in range(3, 8, 2):

    print(x)

print("====================")

The program window is shown in Figure



```
for x in range(5):
    print(x)    #upperbound value 5 will be ignored and it prints from 0 to 4
print ("====================")
#Prints out 3,4,5
for x in range(3, 6):
    print(x)
print("====================")

#Prints out 3,5,7
for x in range(3, 8, 2): #In this statement, the 3rd parameter is the increment given i.e. 2
    print(x) #whereas the default increment is 1.Therefore,it prints 3,5,7.

print("====================")
```

Figure 2.23: Program window of printing the numbers using range() function in for loop

The output of Example 2.12 is shown in Figure

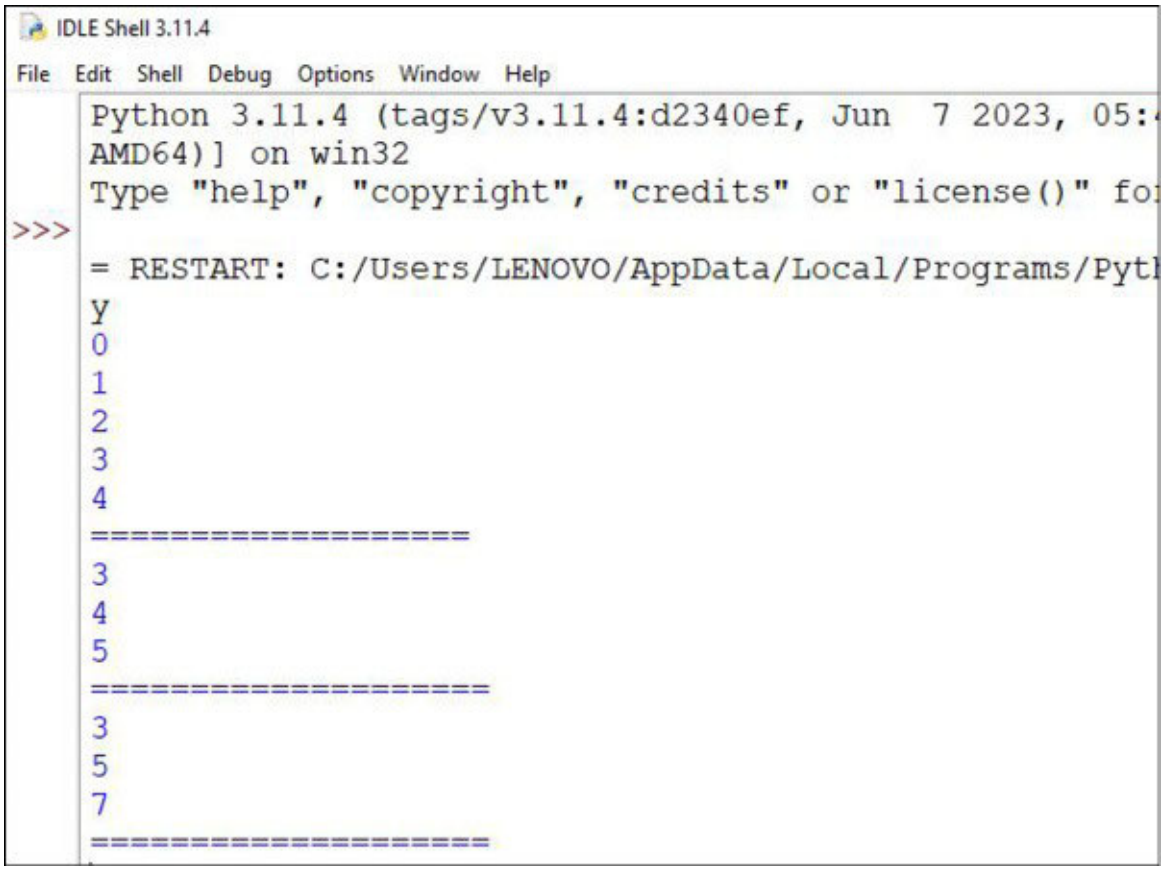


Figure 2.24: Output window of printing the numbers using range() function in for loop

Example 2.13: Write a program to display odd numbers using for loop

In this program, the values are checked in the if condition. If the number is even, the continue statement will return the control to the beginning of the loop and reject all the remaining statements in the current iteration for the loop running.

Code 2.13:

```
# Prints out only odd numbers - 1,3,5,7,9

for x in range(10):

    # Check if x is even

    if x % 2 == 0:

        continue

    print(x)
```

The program window of Example 2.13 is shown in Figure

```
# Prints out only odd numbers - 1,3,5,7,9
for x in range(10):
    # Check if x is even
    if x % 2 == 0:
        continue
    print(x)
```

Figure 2.25: Program window of displaying odd numbers using for loop

The output of the Example 2.13 is shown in Figure

Figure 2.26: Output window of displaying odd numbers using for loop

Example 2.14: Write a program using for loop with arrays

In this program, the list is declared. The len() function is used to return the length of the list, as shown in the program window in Figure In this example, for loop iterate from position in a list declared and range() function is used to iterate through the list and will return a list of the index from 0 to n-1.

Code 2.14:

courses = ['c', 'c#', 'java']

for index in range(len(courses)):

  print 'Current course :', courses[index]

print "Good bye!"

Take a look at the following figure:

```
courses=['c','c#','java']
for index in range(len(courses)):
    print('Current course:',courses[index])

print("Good bye!")
```

Figure 2.27: Program window of using for loop with arrays

The output of Example 2.14 is shown in Figure

Figure 2.28: Output window of using for loop with arrays

Example 2.15: Find the factorial of a number using for loop

In this example, the factorial of a number is calculated using a for loop. A variable fact is initialized to value 1. For loop iterate from 1 to n+1 and factorial is obtained of the number n.

Code 2.15:

```
n = int(input("Enter the Number"))

fact = 1

# Here, the for loop will run from 1 to n+1 i.e. if n = 5,

# then loop will iterate from 1 to 6, but the

# upperbound value is # neglected i.e. loop will

# execute upto 1 to 5 for n = 5

for i in range(1,n+1):

  fact = fact*i

print("The factorial of a",n,"is",fact)
```

The program window is shown in Figure

```
n = int(input("Enter the Number"))
fact = 1
for i in range(1,n+1): #Here,the for loop will run from 1 to n+1 i.e.if n = 5,then loop
    fact = fact*i    # will iterate from 1 to 6, but the upperbound value is
# neglected i.e. loop will execute upto 1 to 5 for n = 5
print("The factorial of a",n,"is",fact)
```

Figure 2.29: Program window of finding a factorial of a number using for loop

The output of the Example 2.15 is shown in Figure

```
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun  7 2023, 05:45:3
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for mo
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/
py
Enter the Number5
The factorial of a 5 is 120
```

Figure 2.30: Output of finding factorial of a number

Example 2.16: Program to find the sum of all numbers stored in a list using for loop

In this example, the sum of all numbers in a list is calculated using a for loop.

Code 2.16:

```
# List of numbers

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum

sum = 0

# iterate over the list

for val in numbers:

    sum = sum+val

# Output: The sum is 48

print("The sum is", sum)
```

The program window is shown in Figure

```
File  Edit  Format  Run  Options  Window  Help
# List of numbers
numbers = [6,5,3,8,4,2,5,4,11]
# variable to store the sum
sum = 0
# iterate over the list
for val in numbers:
        sum = sum+val
# Output: The sum is 48
print("The sum is", sum)
```

Figure 2.31: Program window to find the sum of all numbers stored in a list using for loop

The output of the Example 2.16 is shown in Figure

```
IDLE Shell 3.11.4
File  Edit  Shell  Debug  Options  Window  Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun  7 2023, 05
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" f
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Py
py
The sum is 48
```

Figure 2.32: Output window to find the sum of all numbers stored in a list using for loop

Example 2.17: Finding prime number taking input from the user

In this example, an input number is checked to determine whether it is prime or not. Here, the flag variable is initialized to A for loop is applied using a range function from 2 to n, and within the loop, the number passed as input will be checked in if condition. If the entered number is divisible and the remainder is 0, then the flag variable value becomes that is, the number is not prime, and the loop will break. If the flag variable value is then the number is prime; else, the number is not prime.

Code 2.17:

```
n = int (input("Enter the Number to check"))

flag = True

for i in range(2,n):

    if (n % i) == 0:

        flag = False;

        break

if flag==True:
```

print(n , "is prime")

else:

print(n, "is not a prime number")

The program window of Example 2.17 is shown in Figure

```
n = int(input("Enter the Number to check"))
flag = True
for i in range(2,n):
        if(n%i)==0:
            flag = False;
            break

if flag==True:
            print(n , "is prime")
else:
            print(n,"is not a prime number")
```

Figure 2.33: Example of finding a prime number

The output of the Example 2.17 is shown in Figure

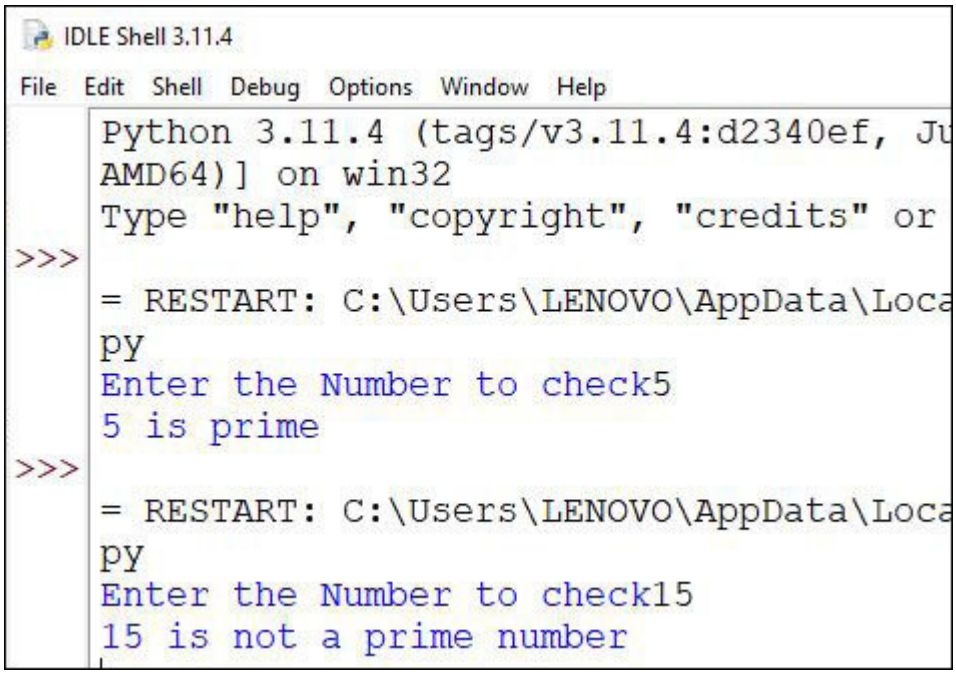


Figure 2.34: Output window of prime number

Break and continue statement

The break and continue statements both work in a loop and can change the flow of a loop. The break statement immediately terminates the loop containing it, and the program will continue after the body of the loop, whereas continue does not terminate the loop and skip the statements inside a loop for the running iteration and continues with the next iteration. The difference between break and continue statements is given in Table

Table 2.1: Difference between break and continue statement

Take a look at the following Figure

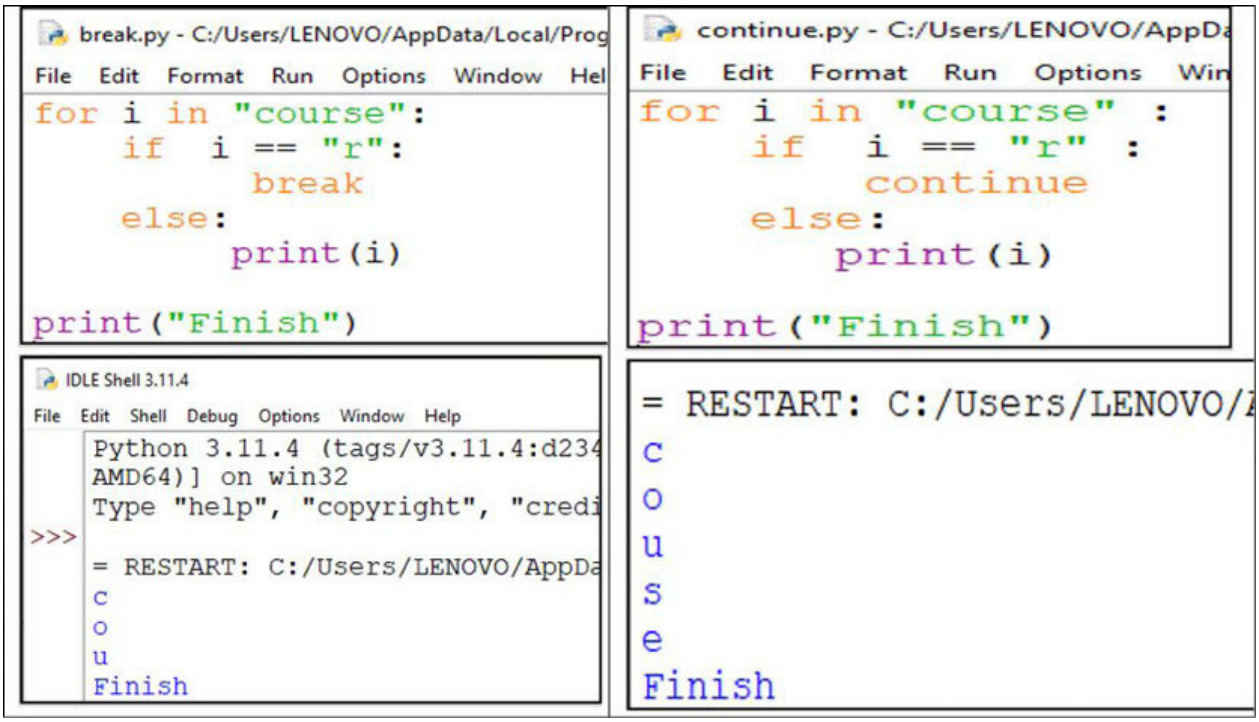


Figure 2.35: Code displaying the difference between break and continue statement

Example 2.18: Write a program to display the prime numbers from 1 to 20 using for loop

In this example, the range() function will return a sequence of numbers and iterate from 1 to 20, and nested for loop is used. Here, for loop will iterate and if the numbers after division have a remainder value of 0, then the loop will break. Otherwise, it will print the prime numbers.

Code 2.18:

```
#to iterate between 1 to 20

for num in range(1,21):

  if num>1:

      for i in range(2,num):

          if (num % i) == 0:

              break

          # else part of the loop

      else:

          print(num)
```

The program window is shown in Figure

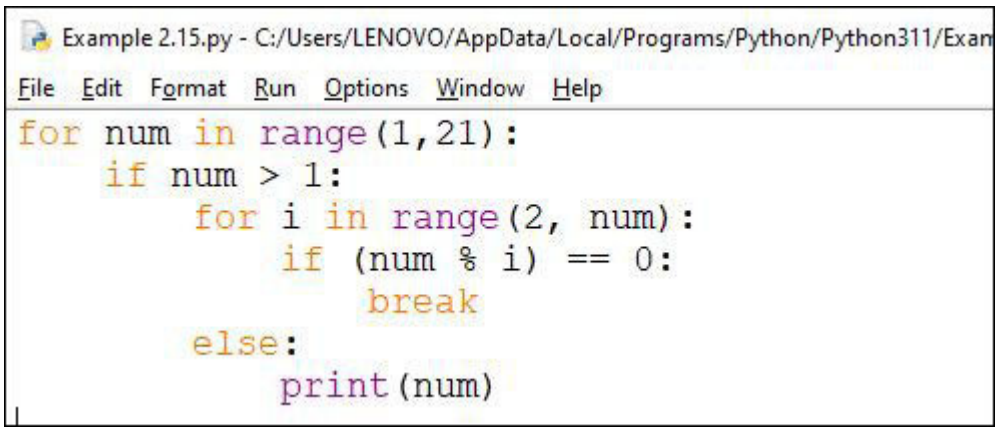


Figure 2.36: Prime numbers from 1 to 20 using for loop

The output of the Example 2.18 is shown in Figure

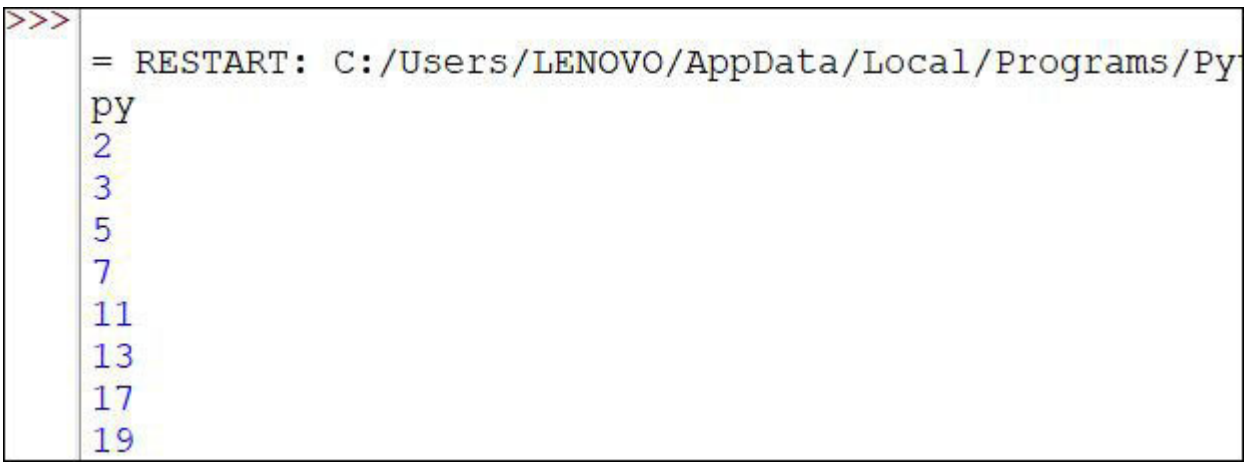


Figure 2.37: Output of prime numbers from 1 to 21

Conclusion

In this chapter, we have learned about the control statements, range() function with the help of examples. The concepts of conditions, loops, range() function, break and continue statements are discussed and illustrated with the help of suitable examples. In the next chapter, we will cover the topics related to arrays and functions.

Questions

Discuss the while and for loop with the help of suitable examples.

What is the range() function? Give an example.

Differentiate between break and continue statements. Give example.

What are control statements? Give a suitable example.

Points to remember

elif in Python is a short form for else if. It is used in conditional statements to check an additional condition if the preceding if condition is false.

A for loop iterates over a sequence (for example, a list, tuple, string, or range) and executes a code block for each item in the sequence.

A while loop is a loop that continues to execute a block of code as long as a specified condition is True.

Break statement is a control statement used to exit a loop prematurely, typically based on a certain condition. In contrast, a continue statement is a control statement used to skip the current iteration of a loop and move to the next iteration.

In nested loops, one loop is placed inside another, creating a loop within a loop.

A range() is a built-in Python function used to generate a sequence of numbers that can be used in loops. It can take up to three arguments: start, stop, and step. Here, start is the starting value of the sequence, stop is the ending value of the sequence and step is the increment between values in the sequence. The default value of the step is 1.

Join our book’s Discord space

Join the book’s Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 3

Arrays and Functions

Introduction

Arrays are the variables having similar data types. Arrays in Python contain values corresponding to the same data types, whereas lists, which are somewhat similar to arrays discussed in the next chapter, contain values corresponding to different data types. Arrays can contain more than one value at a time, and indexing is used to refer to array elements.

A function in Python is a block of a series of statements that carry out a single or several actions. Code is easier to read, comprehend, and maintain when it is divided into usable, modular chunks because of the use of functions. The ability to reuse code is one benefit of utilizing functions.

In this chapter, we will cover both of these topics in-depth!

Structure

In this chapter, the following topics are covered:

Arrays

Functions

Local and global variable scope

Lambda function

Objectives

Upon completing this chapter, you will learn about arrays and functions discussed with the help of suitable examples. You will also know about the local and global scope of variables in a function. Further, the Lambda function utility in Python is discussed with the help of examples of arrays, functions, and lambda functions.

Arrays

An array can have a fixed number of elements. A one-dimensional array, commonly referred to as a single-dimensional array, is a linear array since its members are evaluated in sequential order. Compared to data structures, which have sequential access and only let you access the values they contain in a certain order, arrays provide you random access to items, allowing you to access position elements more effectively. Individuals working with huge datasets can find arrays beneficial for sorting and variable identification tasks.

Example 3.1: Write a program for arrays using append() function

In this example, array a is initialized with three values. Here, the append() function is used to add an element to the last index in the array, and the len() function is used to find the length of an array. format() is a function that returns a value in string format by formatting a k variable value.

Code 3.1:

```
a = [56, 57, 89]

# 56 is at the 0th index

i = a [0]

print (i)

# append() function is used to add element at the

# last in array i.e. at location i = a[3]

a.append(90)

print("The value at 3rd position is", i)

k = len(a)    # find the length of the array
```

print('The length of array is',format(k))

The program window for the above code is shown below:

```
a = [56,57,89]
i = a[0]      # 56 is at the 0th index
print (i)
a.append(90)  # append() function is used to add element at the last in array
i = a[3]
print("The value at 3rd position is",i)
k = len(a)    # find the length of the array
print('The length of array is',format(k))
```

Figure 3.1: Example of an array

The output of the Example 3.1 is shown in Figure

```
>>>
= RESTART: C:/Users/DELL/AppData/Local/Progra
56
The value at 3rd position is 90
The length of array is 4
```

Figure 3.2: Output window of using arrays

Example 3.2: Example of membership operator in

In this program, the Membership operator in is used to test whether the value passed in a variable is found in the given array or not.

Code 3.2:

```
name="sohan"

if name in["akhil","sohan"]:

print("Your name is either vijay lang="hi-IN"> or sohan")
```

The program window is shown in Figure

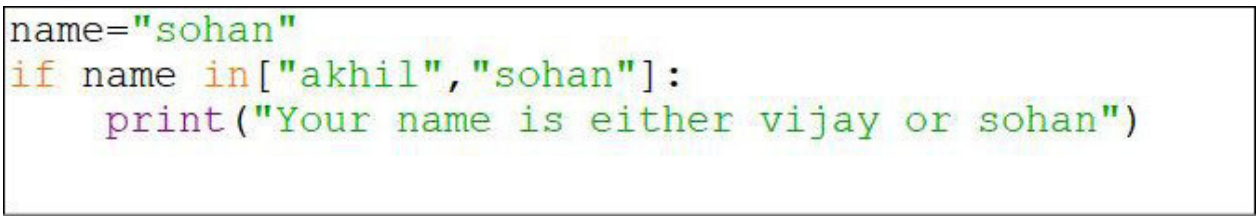


Figure 3.3: Program window of Membership operator in

The output of the Example 3.2 is shown in Figure

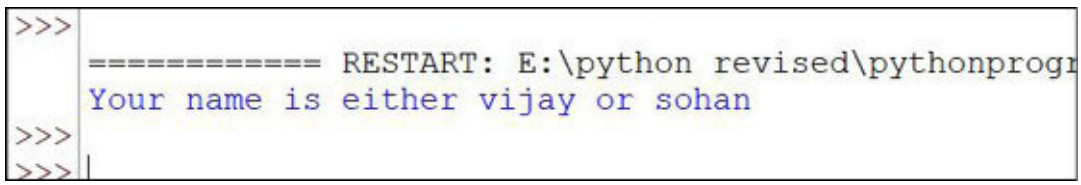


Figure 3.4: Output window of Membership operator in

Example 3.3: Example of in operator and if condition

In this example, an array is initialized, and the value passed in a variable is checked in an array using the if condition and in operator, as shown in Figure When the value is entered incorrectly in the example and is passed in small letters, such as it indicates that the value does not match the value in the array. The message This course is available appears when we enter the right value, since it matches the array:

Code 3.3:

```
a= ["C","C++"]

course=input("Enter the course")

if course in a:

    print("This course is available")

else:

    print("This course is not available")
```

The program window is shown below:

```
a=["C","C++"]
course=input("Enter the course")
if course in a:
    print("This course is available")
else:
    print("This course is not available")
```

Figure 3.5: Program window of in operator and if condition

The output of Example 3.3 is shown in Figure

```
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/
Enter the course:c
This course is not available
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/
Enter the course:C
This course is available
```

Figure 3.6: Output window of in operator and if condition

Example 3.4: Example of len() function in array

In this example, an array is declared as in earlier examples, and the length of the array is obtained using the len() function.

Code 3.4:

```
# array is declared

lst = [23,56,67]

# length of the array is obtained using len() function

val = len(lst)

#  output 3 will be printed

print("The length of an array is", val)
```

The code is shown in the following program window:

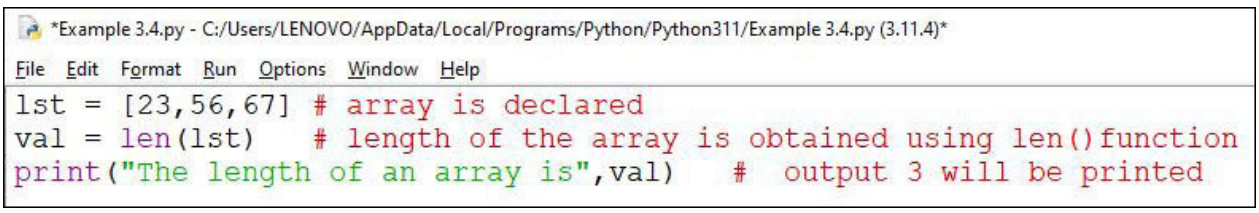

Example 3.4.py - C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/Example 3.4.py (3.11.4)
File Edit Format Run Options Window Help
lst = [23,56,67] # array is declared
val = len(lst)   # length of the array is obtained using len()function
print("The length of an array is",val)   # output 3 will be printed

Figure 3.7: Finding the length of an array

In Example len() function is used to find the length of an array. The output of the Example 3.4 is shown in Figure

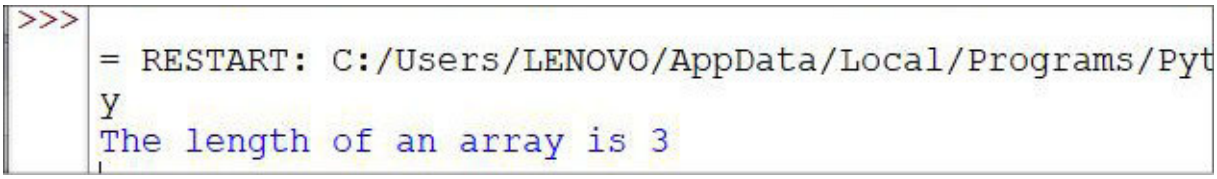


Figure 3.8: Displaying the length of an array

Example 3.5: Display values from 0 to 9 using for loop

In this example, a range() function is used to iterate the numbers from 0 to 9. In this program, the range() function returns the sequence of numbers. The range starts from 0 by default, and the value will be incremented by 1 by default.

Code 3.5:

```
for i in range(10):
    print(i)
```

The program window is shown in Figure

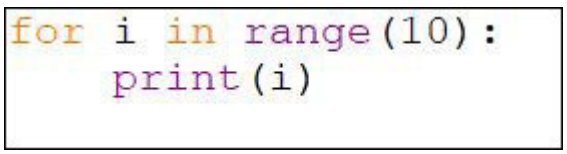


Figure 3.9: Display values from 0 to 9 using for loop and range() function

The output of Example 3.5 is shown in Figure

```
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs
Y
0
1
2
3
4
5
6
7
8
9
```

Figure 3.10: Output using range() function in for loop

Example 3.6: Display values from 0 to 9 using arrays and append() function

In this example, an empty array is initialized, and a for loop using the range() function to iterate from 0 to 9. Within the loop, an append() function is used to add each number to an array.

Code 3.6:

values = [] # initialize an empty array

for i in range(10): #using for loop to populate an array

    values.append(i)

print the array

print(values)

The code is shown in the program window of IDLE in Figure

```
values = []  # initialize an empty array
for i in range(10): #using for loop to populate an array
    values.append(i)

# print the array
print(values)
```

Figure 3.11: Display numbers from 0 to 9 using arrays and an append() function

The output of Example 3.6 is shown in Figure

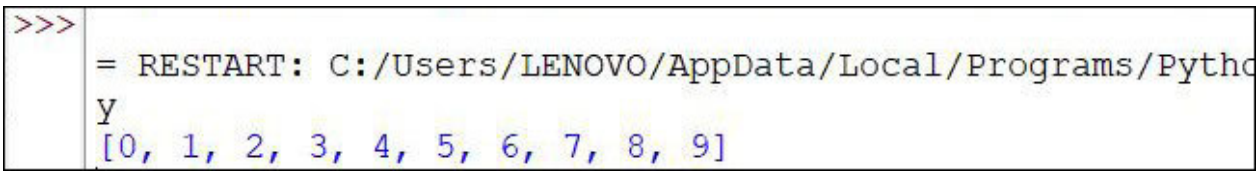


Figure 3.12: Output using append() function in an array

Functions

A function is a block of statements that runs only when called. In functions, data can be passed using parameters, and the value is returned as an output. The advantage of functions is their reusability. The code can be utilized in multiple places in a program by just using the function name. With the help of functions, there is no need to write the statements repeatedly to perform the same task. Functions are powerful constructs in programming that allow you to encapsulate reusable blocks of code and make your code modular and organized.

Note: In Python, ‘def’ keyword is used to define a function.

Example 3.7: Passing parameters in a function

In this program, a function is defined with the name printinfo() having two parameters: name and address, and both are of string datatype.

Code 3.7:

```
def printinfo(name,address):

#  "This prints passed information into this function"

  print("Name: ", name)

  print("Address ", address)

  return

# function printinfo() is called here and address,name are passed as arguments.

printinfo(address="Delhi",name="Mr.ABC")
```

The code is shown in the program window in Figure

```
def printinfo(name,address):
#   "This prints passed information into this function"
   print("Name: ", name)
   print("Address ", address)
   return
# function printinfo() is called here and address,name are passed as arguments.

printinfo(address="Delhi",name="Mr.ABC")
```

Figure 3.13: Example of function using parameters

When the return keyword appears in Python, the function exits and the caller, or the place where the function is called, receives the passed value, like in the example above printinfo(address="Delhi",name="Mr.ABC") function is called. Python functions can have a return statement or not, and a function that doesn't have one generally returns

The output of the Example 3.7 is shown in Figure

```
>>>
    = RESTART: C:\Users\DELL\AppData\Local\Programs\E
    Name:  Mr.ABC
    Address  Delhi
>>>
>>>
```

Figure 3.14: Output window using parameterized functions

Local and global variables in functions

Variables in Python can have a local or global scope. It is referred to as having a local scope when a variable is declared inside a function. Only the function in which it is defined can access it. A variable with a global scope is declared outside of all functions or at the root level of a module. It is accessible inside and outside the function from any point in the code —global variables last through the duration of the entire program. The local variable has priority within the function if the same variable has the same name in both the local and global scopes.

Note: When a variable in Python is declared as global, it can be accessed and changed anywhere, not only inside the current function or code block. This is done by using the global keyword. Rather than creating a new local variable with the same name, using global before a variable inside a function tells Python that the variable is defined in the global scope as shown in Example 3.9.

Example 3.8: Local and global variable

In this program, the global variable total is declared, and inside the function, the local variable total is declared. The local variable has the scope inside the function. It can be used inside the function only, whereas the global variable has its scope outside the function.

Code 3.8:

```
# Global variable is declared

total = 0;

def sum(arg1,arg2):      # function sum() is defined here

  total = arg1 + arg2;  #  local variable is declared

print ("Inside the function local total:",

  return total;

sum(10,20); # function is called here

print the function global total : ",
```

The program window is shown in Figure

```
File  Edit  Format  Run  Options  Window  Help
total = 0;                  #Global variable is declared
def sum(arg1,arg2):         #function sum() is defined here
   total = arg1 + arg2;     #local variable is declared
   print("Inside the function local total :",total)
   return total;
sum(10,20);          #function is called
print ("Outside the function global total :",total)
```

Figure 3.15: Example of local and global variable

The output of the Example 3.8 is shown in Figure

```
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/Example3.7.py
Inside the function local total : 30
Outside the function global total : 0
```

Figure 3.16: Output of global and local variable

Example 3.9: Write a program for a variable having global scope

In this example, a global variable var1 is defined. A function named global_fun() is defined, and the var1 value is modified using the global keyword.

The program in the IDLE window is shown in Figure

```
var1 = 25  # This is a global variable

def global_fun():
# This allows to modify the global variable 'var1' within the function
    global var1
    var1 = var1 + 5
    print(var1)

#function is called here
global_fun()  # Output: 30
# Output: 30, as the global variable 'var1' has been modified within the function
print(var1)
```

Figure 3.17: Variable modified using global keyword

The output of Example 3.9 is shown in Figure

```
>>>
========== RESTART: E:/python revised/pythonprograms/
30
30
>>>
>>>
```

Figure 3.18: Output window of global variable

Example 3.10: Write a calculator program using the concept of functions

In this program, the menu() function is defined with five options, just like in switch case statements. On pressing option 1, the add() function for addition is called; for option 2, the sub() function is called for subtraction, similarity other options are added. Here, while loop is executed till it is true and inside it, a menu() function is called. When the option is selected by the user, mul() and div() functions are called accordingly. Inside these functions, parameters are passed.

Code 3.10:

```
menu():

    print ("your options are:")

    print (" ")

    print ("1 Addition")

    print ("2 Subtraction")

    print ("3 Multiplication")

    print ("4 Division")
```

```
    print ("5 Quit ")

    print (" ")

    return input ("Choose your option: ")


# this adds two numbers given

def add(a,b): #  add()  function is defined

    print (int(a), "+",int(b), "=",int(a) +int(b))

# this subtracts two numbers given

def sub(a,b): # sub() function is defined

    print (int(a), "-",int(b), "=",int(a) – int(b))

# this multiplies two numbers given

def mul(a,b): # mul() function is defined

    print (int(a), "*",int( b), "=",int(a) * int(b))

# this divides two numbers given

def div(a,b): # div() function is defined
```

```
    print (int(a), "/",int( b), "=",int(a) /int( b))

loop = 1

choice = 0

while loop==1:

# menu () function is called here

    choice = menu()

    if choice == '1':

# inputs from user are passed as  parameter in function


        add(input("Add this:"),input("to this:"))

    elif choice == '2':

        sub(input("Subtract this:"),input("from this:"))

    elif choice == '3':

        mul(input("Multiply this:"),input("by this:"))
```

```
    elif choice == '4':

        div(input("Divide this:"),input("by this:"))

    elif choice == '5':

        loop = 0

print( "Thankyou !")
```

The program window of Example 3.10 in IDLE is shown in Figure

```
def menu():
    print("your options are:")
    print("")
    print ("1 Addition")
    print ("2 Subtraction")
    print ("3 Multiplication")
    print ("4 Division")
    print ("5 Quit")
    print (" ")
    return input("Choose your option:")
# this adds two numbers given
def add(a,b):           #  add()  function is defined
    print(int(a), "+", int(b), "=", int(a)+int(b))
# this subtracts two numbers given
def sub(a,b):           # sub() function is defined
    print (int(a), "-", int(b), "=", int(a) - int(b))
# this multiplies two numbers given
def mul(a,b):           # mul() function is defined
    print (int(a), "*", int(b), "=", int(a) * int(b))
# this divides two numbers given
def div(a,b):           # div() function is defined
    print (int(a), "/", int(b), "=", int(a) / int(b))

loop = 1
choice = 0
while loop==1:
    choice=menu()       # menu() function is called here
    if choice=='1':
        # inputs from user are passed as parameter in function
        add(input("Add this: "),input("to this: "))
    elif choice=='2':
        sub(input("Subtract this: "),input("from this: "))
    elif choice=='3':
        mul(input("Multiply this: "),input("by this: "))
    elif choice=='4':
        div(input("Divide this: "),input("by this: "))
    elif choice=='5':
        loop = 0
print ("Thankyou !")
```

Figure 3.19: Program window of calculator using functions

The output of Example 3.10 is shown in Figure

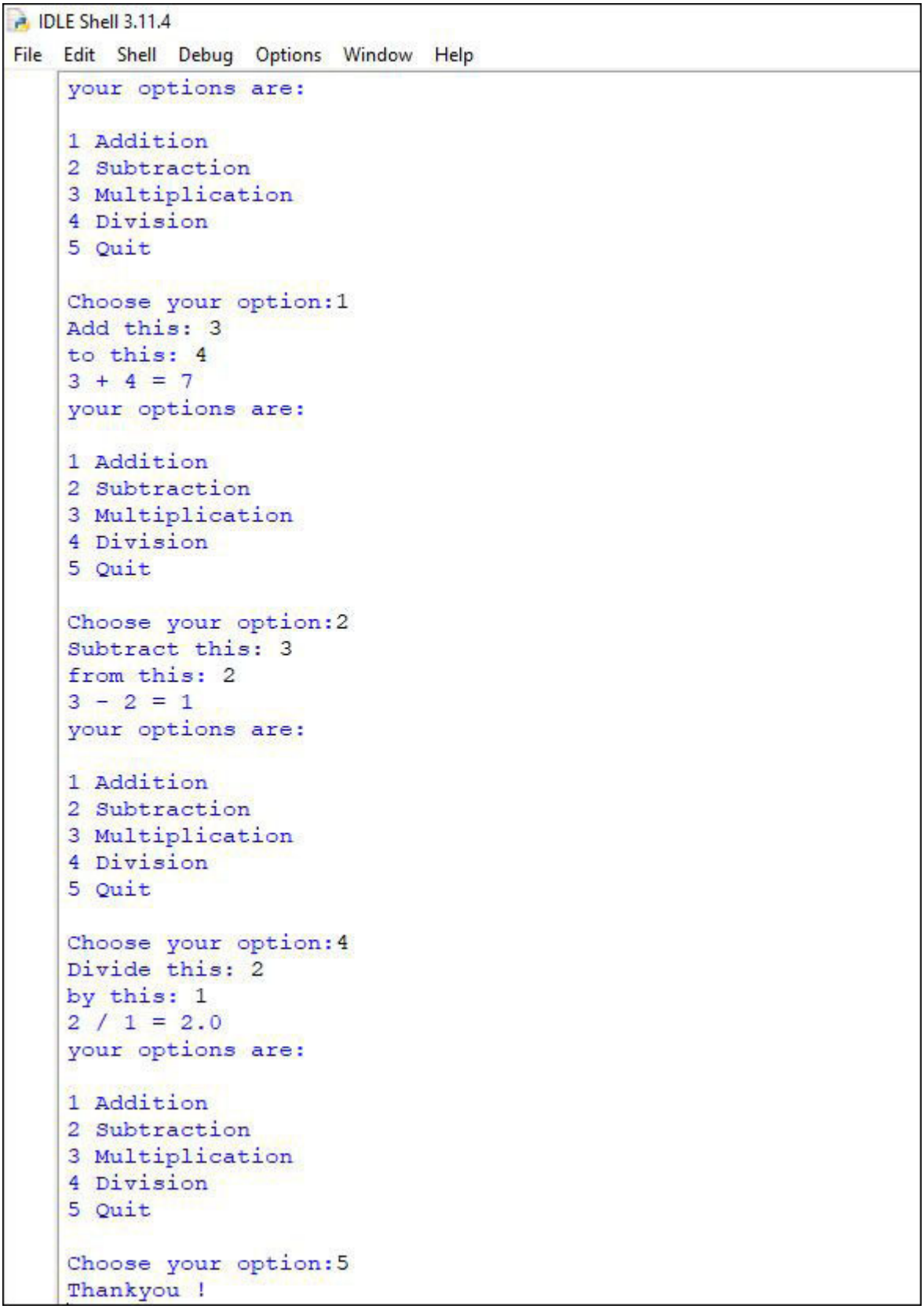


Figure 3.20: Output of the calculator program

Example 3.11: Function returning the absolute value of a number

In this example, a function is defined, and it will return the absolute value of a number passed as a parameter inside the function.

Code 3.11:

```
def absval(num):

# """This function returns the absolute value of the entered number"""

if num >= 0:

    return num

else:

    return -num

print(absval(2))

print(absval(-4))
```

The program window is as shown in Figure

```
def absval(num):
    # """This function returns the absolute value of the entered number"""
        if num >= 0:
                return num
        else:
                return -num
print(absval(2))
print(absval(-4))
```

Figure 3.21: Program window of a function returning the absolute value of a number.

The output of the Example 3.11 is shown in Figure

Figure 3.22: Output window of finding an absolute value of a number using function

Example 3.12: Write a program to sort array elements

In this program, the built-in sort() function is used to sort the elements of an array. The program window in the IDLE shell is shown in Figure

```
n = [15,1,10,45]
n.sort()
print(n)
```

Figure 3.23: Sort array elements

Code 3.12:

n = [15, 1, 10, 45]

n.sort()

print (n)

The output of the Example 3.12 is shown in Figure

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:4
Type "help", "copyright", "credits" or "license()" for
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Pyth
[1, 10, 15, 45]

Figure 3.24: Sorting of elements

Example 3.13: Write a program to find the square of a number using functions

In this example, a function square is defined, and it will return the square of a number that is being input as a parameter inside the function.

Code 3.13:

```
def square(num):

    return num ** 2

n = int(input('Enter a number'))

print('The square of a number is:',square(n))
```

The program window in the IDLE shell is shown in Figure

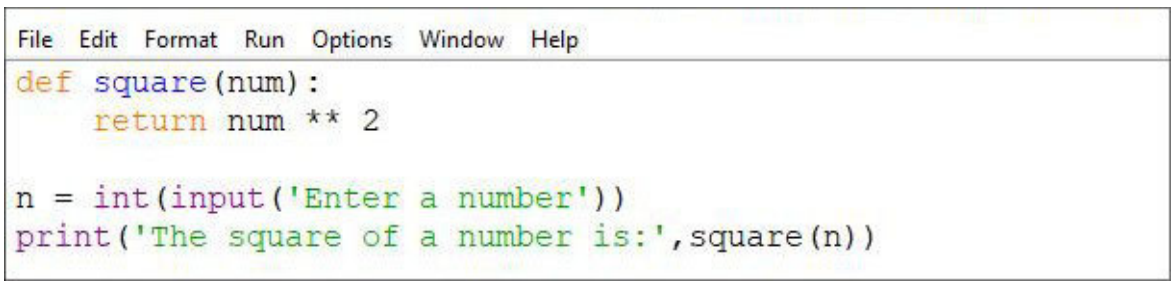

File Edit Format Run Options Window Help
def square(num):
return num ** 2
n = int(input('Enter a number'))
print('The square of a number is:',square(n))

Figure 3.25: Finding the square of a number using the function

The output of Example 3.13 is shown in Figure

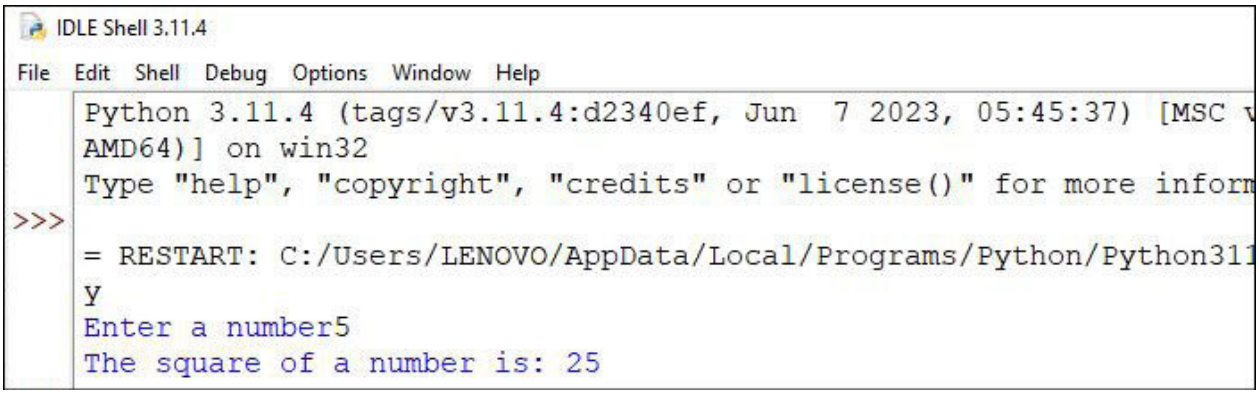

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more inform
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python311
y
Enter a number5
The square of a number is: 25

Figure 3.26: Output of finding the square of a number using function

Lambda function

A lambda function is also known as an anonymous function. These functions are inline and are small blocks of code that can be defined without a name for performing simple operations.

Example 3.14: Adding numbers using the lambda function

In this example, a lambda function is defined, and the addition operation is performed without defining a separate function.

Code 3.14:

lambda function to add two numbers

n = lambda x, y: x + y

Using the lambda function

result = n(13, 15)

print('Addition of x and y is', result)

The program window is shown in Figure

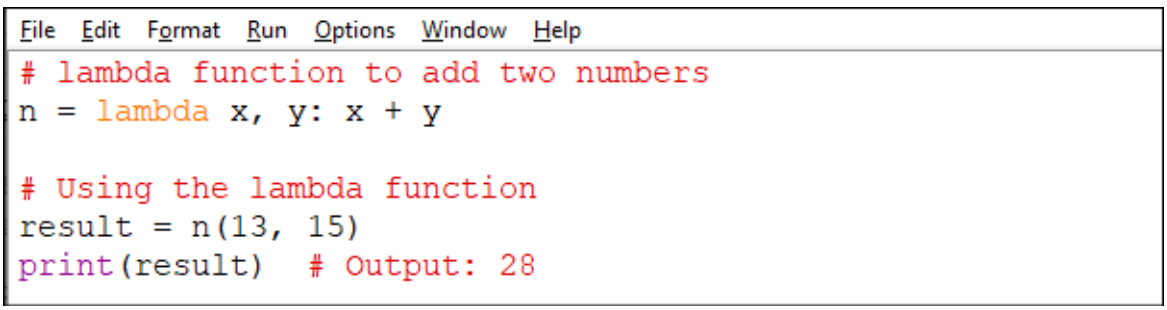

File Edit Format Run Options Window Help
# lambda function to add two numbers
n = lambda x, y: x + y
# Using the lambda function
result = n(13, 15)
print(result) # Output: 28

Figure 3.27: Lambda function to add two numbers

The output of Example 3.14 is shown in Figure

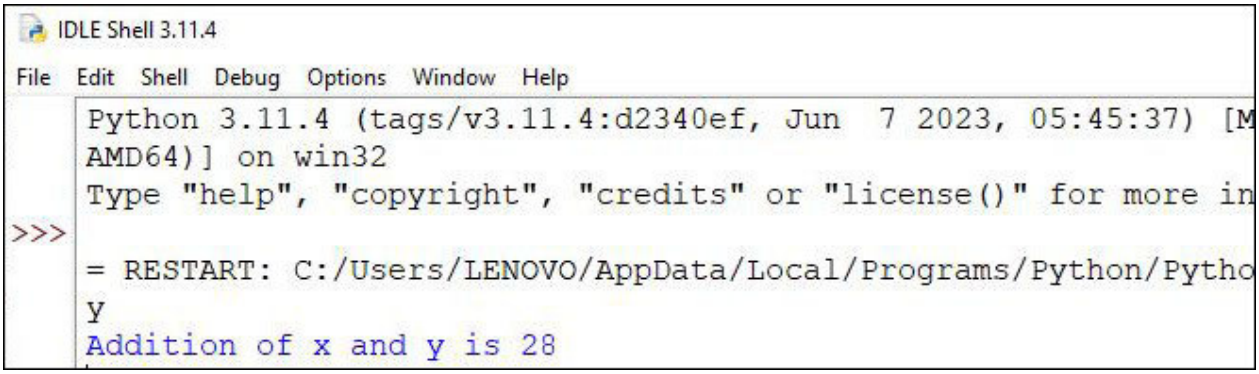

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [M
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more in
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Pytho
y
Addition of x and y is 28

Figure 3.28: Output using Lambda function

Conclusion

In this chapter, we learned about arrays with the help of illustrative examples. An important topic of functions is covered, followed by simple examples. The local and global variable scope within the functions was discussed, and topics related to the Lambda Function were covered. In the next chapter, we will discuss the fundamentals of lists, tuples, iterators, and generators.

Points to remember

Arrays are the variables having similar data types.

Functions in Python are defined using the def keyword.

Functions return value using the return statement. When the return keyword appears in Python, the function exits and the caller, or the place where the function is called, receives the passed value. Python functions can have a return statement or not, and a function that does not have one generally returns None value.

The variables used in the function definition to indicate the expected input values are referred to as parameters, sometimes known as formal parameters in functions. The actual values, often referred to as arguments, that will be provided when the function is called are simply substitutions or placeholders for the values that are termed parameters.

A lambda function is also known as an anonymous function. These functions are inline functions and are small blocks of code that can be defined without a name for performing simple operations like a square, a cube of a number, and so on.

A variable defined at the top level of a program or within a module is referred to as a global variable since it can be accessed from anywhere

within that program or module. They can be called from anywhere in the code, including from other functions, classes, or modules, because they have a global scope.

When a variable in Python is declared as global, it can be accessed and changed anywhere, not only inside the current function or code block. This is done by using the global keyword.

A local variable in a function is a variable that is declared and used within the scope of that particular function. Local variables are only available and useful within the block of code in which they are declared; hence, they have a restricted scope.

Exercises

What do you understand by arrays?

What are functions? Explain with the help of a suitable example.

Differentiate between local and global variable scope.

What is the Lambda function? Give example.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 4

Lists, Tuples, Iterators Generators, and Sets

Introduction

In this chapter, we will discuss the concept of lists, tuples, iterators, and generators. Although all of these are data structures or constructs that are used to keep, iterate through, and modify collections of data, they each have distinctive characteristics.

Structure

In this chapter, the following topics will be covered:

Lists

Slicing

Tuples

Iterators and generators

Sets

Objectives

By the end of this chapter, learners will be able to understand the difference between lists and tuples. Apart from that, we will learn about the concept of iterators and generators with the help of suitable examples.

Lists

In Python, lists are used just like arrays in order to store the data. Python does not have built-in support for arrays.

In Python, a list is a built-in data structure and a commonly used data type. Lists can contain items of multiple data types and are By mutable, we mean to say that the content of lists can be changed. The elements can be added, removed, and modified after their creation. Lists are used to hold an ordered sequence of elements. They can be accessed via positive or negative indexing.

Note: Arrays are used for storing large amounts of data and work efficiently for arithmetic calculations in comparison to lists. The difference between arrays and lists is that lists are not homogeneous, whereas arrays are homogenous. The lists can store multiple data at the same time.

Example 4.1: Example of using the append() function in lists

In this example, an empty list is created, and the append() function is used to add elements to the list. Here, elements in the list are displayed using a print statement and through for loop.

Code block 4.1:

```
mylist = [] # empty list is created

# append() function is used to add elements into list

mylist.append(1)

mylist.append(2)

mylist.append(3)

print(mylist[0]) # prints 1

print(mylist[1]) # prints 2

print(mylist[2]) # prints 3

# prints out 1,2,3

for x in mylist:      # for loop is used
```

print(x)

The program window is shown in the Figure

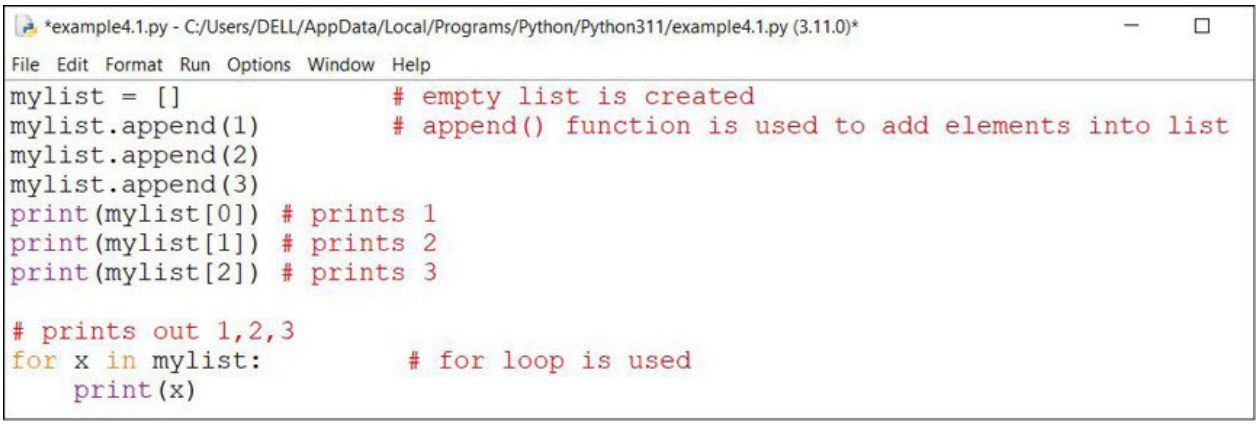

example4.1.py - C:/Users/DELL/AppData/Local/Programs/Python/Python311/example4.1.py (3.11.0)
File Edit Format Run Options Window Help
mylist = [] # empty list is created
mylist.append(1) # append() function is used to add elements into list
mylist.append(2)
mylist.append(3)
print(mylist[0]) # prints 1
print(mylist[1]) # prints 2
print(mylist[2]) # prints 3
prints out 1,2,3
for x in mylist: # for loop is used
print(x)

Figure 4.1: Program window of Example 4.1

The output of Example 4.1 is shown in Figure

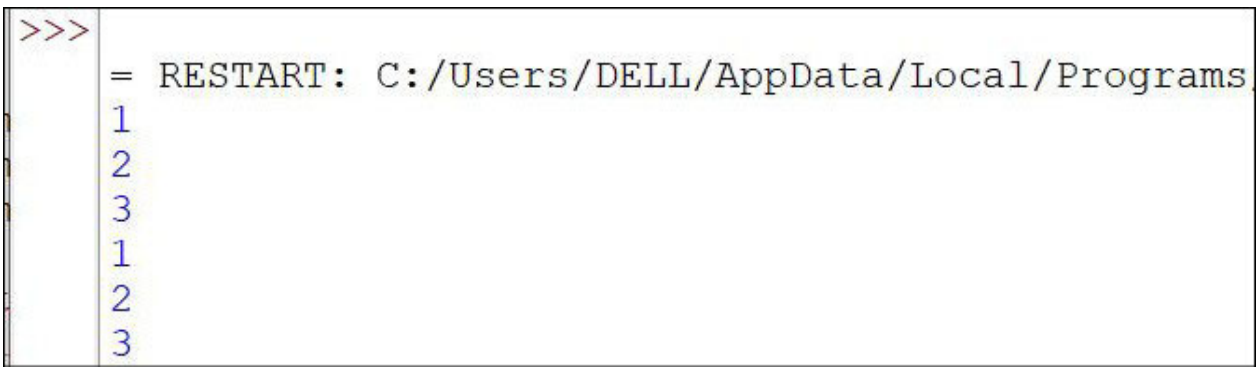


Figure 4.2: Output of lists

Slicing

The process of extracting a segment of a sequence using a certain range of indices from a list, tuple, or string is known as slicing in Python. The standard syntax of slicing is where start is the index of the first element to include and stop is the index where the slicing stops. However, the element at the stop index is not included in a slice. As an additional option, we can use sequence[start:stop:step] to indicate the step size. Here, step size means the interval between indices and it determines how many indices to skip between elements. Here is an illustration of a slice as mentioned below:

```
a = [11,31,51,62,72,82,93]

slicedlist = a[1:5:2]

print(slicedlist)
```

In a statement slicedlist = element at the index is 31 and after getting a step size of two that is increment of two, 62 will be displayed. However, element at the index will not be included in a slice.

Example 4.2: Example of slicing a list and for storing multiple data items

In this example, list1 and list2 are created of multiple data items. list1 stores strings as well as integer values, whereas list2 stores integer values as shown in the IDLE window in Figure In this example, slicing is done in the list. Similarly, array slicing is also done to extract particular values from an array.

Code 4.2:

list1 = ['physics', 'chemistry', 2018, 2019];

list2 = [1, 2, 3, 4, 5, 6, 7];

print "list1[0]: ", list1[0]

print "list2[1:3]: ", list2[1:3]

Take a look at the following figure:

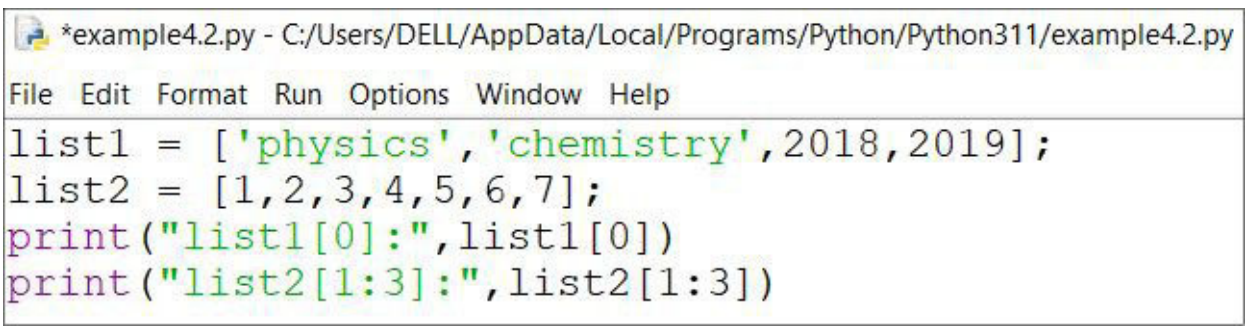


Figure 4.3: Program window of slicing a list

The output of Example 4.2 is shown in Figure The output shows that list1[0] will display physics as the output because at index, physics is the value, and list2[1:3] will display the values at index 1 and 2, that is, 2 and 3 is the output. The upper bound value will be ignored, that is, the index value. Take a look at the following figure that displays the output:

Figure 4.4: Output of slicing a list

Note: operator is used to slice a list; that is, we can take out a particular portion of a list.

Double colons (::) can be used in Python to build a slice with additional slicing. With expanded slicing, you may specify a step value in addition to the start and finish indices. The following syntax is for additional slicing with two colons:

Note: List [start: stop: step]. In this syntax, ‘start’ represents the index from where the slice starts, ‘stop’ is the index where the slice ends, and ‘step’ size represents the number of elements to skip between each selected element. The step size may be zero, positive, or negative value.

Some of the code blocks explaining the meaning of “:” are mentioned in Example

Example 4.3: Example of Slicing using double colon for list sequences

In this example, a list is defined for numerical values and strings. To get every element in a list defined, [::2] syntax will be used, whereas for getting every element in a list defined, [::3] syntax will be used. In a list, the [::-1] syntax represents the reverse order of elements in a list.

Code 4.3:

```
# Here a list is defined

lst1=[7,8,9,10,11,12,15]

# Get every second element

output = lst1[::2]

print(output)  # Output: [7,9,11,15]

# Get every third element

output = lst1[::3]

print(output)  # Output: [7,10,15]

# In case of Strings
```

```
lst2= "Python Course"

# Start from 1st index in above given string

# and get every second character


output= lst2[1::2]

print(output)  # Output: yhnCus

# Reverse of a List1 using negative step size i.e.-1

output = lst1[::-1]

print(output)

# Reverse of a String using negative step

output = lst2[::-1]

print(output)

# Following the start,stop and step sequence for List

# Here, it means start from 0th index,
```

```
# 4th index is the stop i.e.slice end and

# 2 is the step size

lst3 = [10,11,12,15,25,27]

output = lst3[0:5:2]

print(output)
```

The program window of Example 4.3 is shown in Figure

```
# Here a list is defined
lst1=[7,8,9,10,11,12,15]
# Get every second element
output = lst1[::2]
print(output)  # Output: [7,9,11,15]
# Get every third element
output = lst1[::3]
print(output)  # Output: [7,10,15]
# In case of Strings
lst2= "Python Course"
# Start from 1st index in above given string
# and get every second character
output= lst2[1::2]
print(output)  # Output: yhnCus
# Reverse of a List1 using negative step size i.e.-1
output = lst1[::-1]
print(output)
# Reverse of a String using negative step
output = lst2[::-1]
print(output)
# Following the start,stop and step sequence for List
# Here, it means start from 0th index,
# 4th index is the stop i.e.slice end and
# 2 is the step size
lst3 = [10,11,12,15,25,27]
output = lst3[0:5:2]
print(output)
```

Figure 4.5: Program window of Slicing using Double Colon (::)

The output window is shown in Figure

```
>>>
    = RESTART: C:/Users/LENOVO/AppData/Local
    g.py
    [7, 9, 11, 15]
    [7, 10, 15]
    yhnCus
    [15, 12, 11, 10, 9, 8, 7]
    esruoC nohtyP
    [10, 12, 25]
```

Figure 4.6: Output window of slicing using double colons (::)

Tuples

A tuple is a sequence of immutable Python objects. Tuples are sequences, just like lists. The differences between tuples and lists are the tuples cannot be changed, unlike lists and tuples use parentheses '()', whereas lists use square brackets '[ ]'.

Note: In tuples, once a tuple is created, we cannot change its elements, which is the major difference between tuples and lists in Python.

Creating a tuple is as simple as putting different comma-separated values. Optionally, you can put these comma-separated values between parentheses also.

Example 4.4: Example for concatenation of tuples

In this example, a tuple is declared, and tuple1 and tuple2 values are concatenated.

Code 4.4:

```
tup1 = (12, 34.56);

tup2 = ('abc', 'xyz');

# Following declaration is not valid for tuples

# tup1[0] = 100;

# So let's create a new tuple as follows

tup3 = tup1 + tup2;

print (tup3)
```

The program window in IDLE is shown in Figure

```
tup1 = (12,34.56);
tup2 = ('abc', 'xyz');
#Following declaration is not valid for tuples
#tup1[0] = 100;
#So let's create a new tuple as follows
tup3 = tup1 + tup2;
print(tup3)
```

Figure 4.7: Example of tuples

The output of the Example 4.4 is shown in Figure

```
>>>
= RESTART: C:/Users/DELL/AppData/Local/Programs/Pythor
(12, 34.56, 'abc', 'xyz')
```

Figure 4.8: Output of tuples

Example 4.5: Nested tuples

In Example a nested tuple is created, and elements are accessed similar to 2-dimensional arrays in a matrix.

Code 4.5:

Nested tuples

tup=((1, 2),(3,4),(5,6))

#Access the 2nd row and first element

print('Element at 2nd row and 1st element is',tup[2][1])

The program window in the IDLE environment is shown in Figure

```
# Nested tuples
tup=((1, 2),(3,4),(5,6))
#Access the 2nd row and first element
print('Element at 2nd row and 1st element is',tup[2][1])
```

Figure 4.9: Nested tuples

The output window of Example 4.5 is shown in Figure

Figure 4.10: Output window of Example 4.5

Python iterators

Iterator means repetition, and they are simplified using loops. An Iterator in Python is an object that is iterated upon to iterate through all the values given.

The two most important methods for iterators are:

__iter__()

__next__()

Iterators and iterable objects are implemented in Python using the __iter__() and __next__ () functions. These functions are a component of the iterator protocol, which enables the use of for loops and other iteration structures to iterate over objects. When an iterator object for an iterable is generated, the __iter__() function is invoked. A __next__() or __iter__() method that returns the subsequent value in the iteration must exist for this object. To obtain the subsequent value in the loop, the __next__() method is invoked. Iterators are effectively used with for loop and in lists, tuples, and strings.

Example 4.6: Write a program to print a list of courses using iterators

In this program, a list is created, and the next() function is used to return the elements in an iterator.

Code 4.6:

```
#  list is created

course_list = ["java","dotnet","python"]

course_list = iter(course_list)

print(next(course_list))

print(next(course_list))

print(next(course_list))
```

The program window in the IDLE shell is shown in Figure

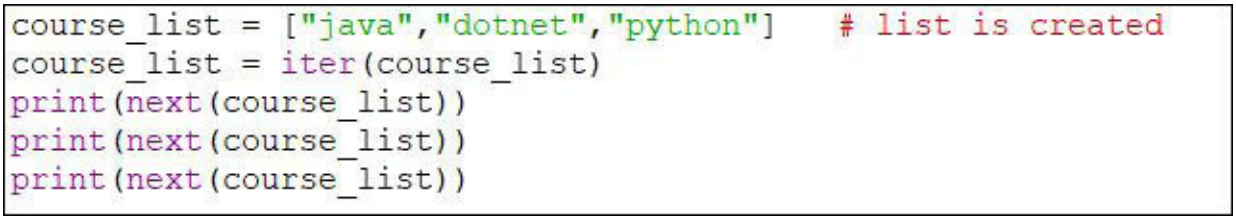


Figure 4.11: Program window of iterators

The output of the Example 4.6 is shown in Figure

Figure 4.12: Output of iterators

Example 4.7: Write a program using iterators through loops

Iterators are an essential part of Python's iteration and looping mechanisms. In Example iterators are used in for loop to iterate over elements in a data structure, that is, lists. The advantage of using iterators is that it allows us to traverse through a sequence of values, one at a time, without loading the entire sequence into memory.

Code 4.7:

```
course_list = ["java","dotnet","python"]

for i in course_list:

  print(i)
```

The program window of Example 4.7 is shown in Figure

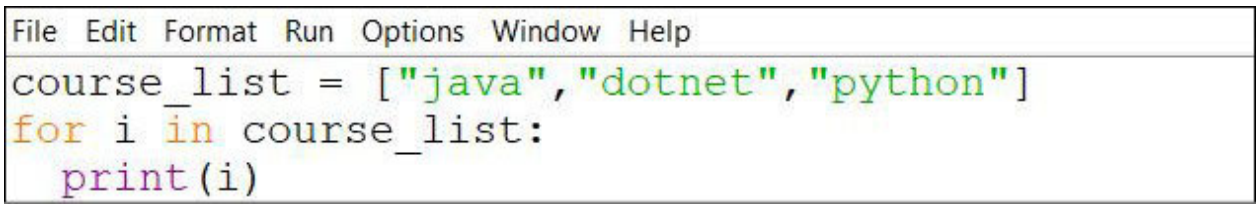


Figure 4.13: Iterators through loops

The output of the Example 4.7 is shown in Figure

```
>>>
= RESTART: C:\Users\DELL\AppData\Local\Prog
java
dotnet
python
```

Figure 4.14: Output of iterators using loops

Python generators

Generators are the simplest way of creating iterators. Python generators are just like a function where the function is defined separately using the def keyword. It returns an object, which is the iterator itself.

The advantage of using generators is that it simplifies the program and can be reused many times. Python generators allow us to create iterators in a more concise and memory-efficient manner compared to using custom iterator classes. They generate a sequence of values and yield values one at a time without storing the entire sequence in memory. It makes generators memory efficient, especially for large sequences, because generators generate values as needed and do not store the entire sequence in memory. Generators use a function-like syntax with the yield keyword instead of defining _ _ iter_ _() and _ _next_ _() methods.

Note: In generators, in place of the return statement, the “yield” statement is used.

Example 4.8: Write a program for generators

In this example, a separate generator is defined with the name course_generate() and is saved with the name

Code 4.8:

```
def course_generate():
    yield("C")
    yield("C++")
    yield("Java")
yield("Python")
    yield("Php")
    yield("Vb.Net")
    yield("Asp.Net")
yield("Android")
```

The program window is shown in Figure

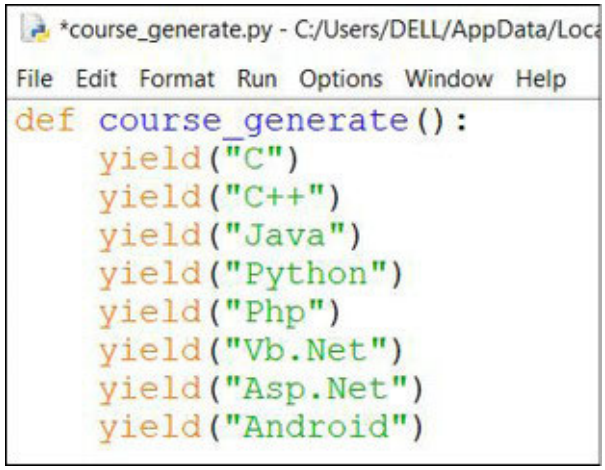


Figure 4.15: course_generate.py

Note: Save this function as 'course_generate.py'. It is for the understanding of learners that we will implement 'course_generate' in the file 'generate1.py' show in Figure Since it is a user-defined function, we are free to give it any name we choose. However, we must implement it correctly wherever we use it and take care of not making mistakes while invoking the function.

Now, this generator will be called in another Python program saved with the name as shown in Figure Here, the import statement is used to import the module named

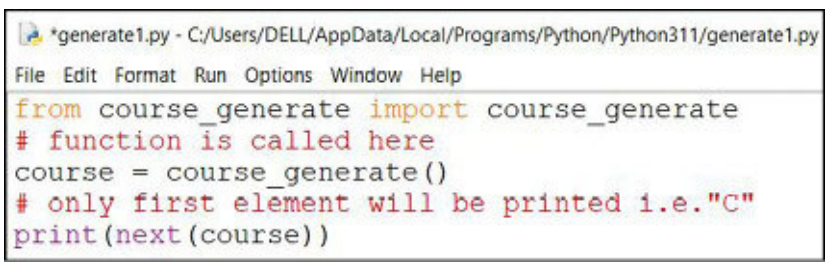

*generate1.py - C:/Users/DELL/AppData/Local/Programs/Python/Python311/generate1.py
File Edit Format Run Options Window Help
from course_generate import course_generate
# function is called here
course = course_generate()
# only first element will be printed i.e."C"
print(next(course))

Figure 4.16: course_generate () function is called in generate1.py program

Here, for instance, the keyword from is used to import particular objects into the current program, such as pre-defined functions from a module or package, all within a single statement.

Code 4.9:

```
from course_generate import course_generate

# function is called here

course = course_generate()

# only first element will be printed i.e. "C"

print(next(course))
```

The output of the Example 4.8 is shown in Figure 4.17 below:

```
>>>
== RESTART: C:/Users/DELL/AppData/Local/Programs/Python/Python311/generate1.py
C
```

Figure 4.17: Output of generator

Example 4.9: Simplify the Example 4.8 using loops

In this example, the same function course_generate is used, but instead of displaying one value at a time, all values defined in course_generate() function are displayed using for loop.

Code 4.10:

```
from course_generate import course_generate

course = course_generate()

for i in course:

print(i)
```

The program window of generate2.py using IDLE is shown in Figure

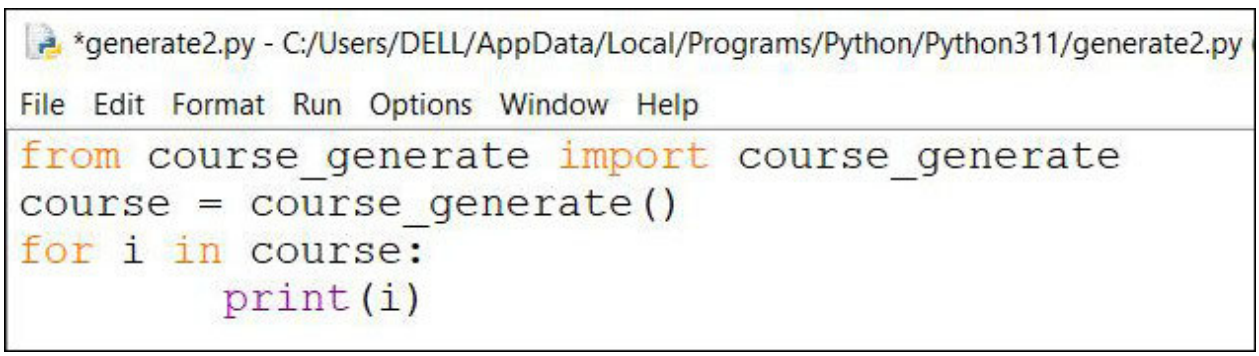


Figure 4.18: generate2.py program window

The output of Example 4.9 is shown in Figure 4.19 below:

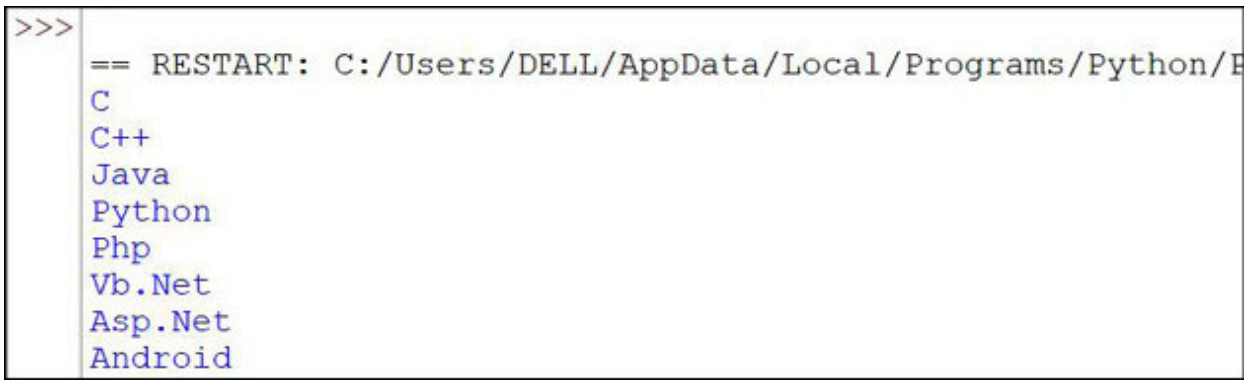


Figure 4.19: Output window of generate2.py

Sets

An unordered group of distinct elements is called a set in Python. It is a data type that the language comes with built-in. Similar to lists or dictionaries, sets are distinguished by the fact that duplicate elements are not allowed and element order is not ensured. In Python, you can use the set() constructor or the curly braces {} to build a set.

Example 4.10: Example of a set and its operations

In this example, a set is created using curly braces in set1 and set2 is created using set() constructor. The operations like union, intersection, and difference are implemented on the sets. It is noteworthy that sets are unordered and do not support slicing or indexing. Use a list or tuple if you need an ordered collection with distinct elements. Sets can have elements added or removed from them because they are changeable that is mutable.

Code 4.11:

```
set1 = {11,21,33,44,53}

set2 = set([35,44,56,67,75])

union = set1|set2

print(union)

intersection = set1&set2

print(intersection)

difference = set1-set2

print(difference)
```

set1.add(68)

print(set1)

set2.remove(35)

print(set2)

The program window using IDLE is shown in Figure

```
set1 = {11,21,33,44,53}
set2 = set([35,44,56,67,75])
union = set1|set2
print(union)
intersection = set1&set2
print(intersection)
difference = set1-set2
print(difference)
set1.add(68)
print(set1)
set2.remove(35)
print(set2)
```

Figure 4.20: Program window of set and its operations

The output of Example 4.10 is shown in Figure

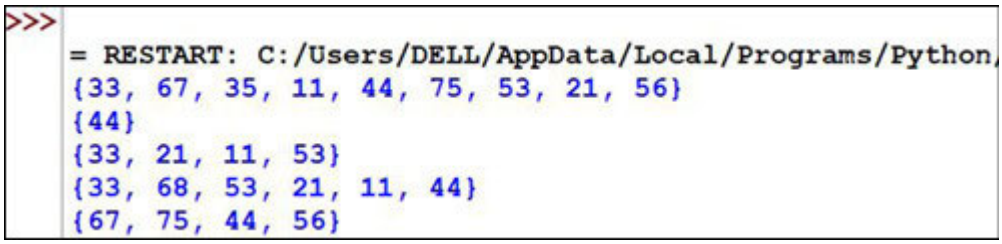


Figure 4.21: Output window of set and its operations

Conclusion

In this chapter, lists and tuples are covered with the help of suitable examples. The concept of slicing is well explained in detail. Apart from that, Python iterators and generators and set collection types are discussed. In the next chapter, learners will learn the concept of dictionaries and modules.

Points to remember

The common element of lists and tuples is that tuples are also the ordered collection of elements. Tuples also contain elements of heterogeneous data types, like in the case of lists, but tuples are immutable. By immutable, we mean the elements, once created, cannot be changed.

An iterator is an object that implies a data sequence and enables us to navigate it.

Iterators are commonly used with a for loop to iterate over elements. The advantage of using iterators is that they help in reducing memory usage.

The __iter__() and __next__() are the methods of iterators.

Generators are also a type of iterator that allows us to iterate over a large data sequence without storing it in memory all at once.

Generators are defined using functions, and yield is the keyword used. It preserves the function state and enables us to continue from where we left.

In Python, double colons (::) are used for enhanced slicing, and it is a flexible method to extract sequence subsets while enabling step size.

Exercises

Differentiate between lists and tuples.

Discuss some of the pre-defined functions of lists. Give a suitable example.

What are nested tuples?

Discuss Python iterators and generators with the help of suitable examples.

What do you mean by slicing? Give an example of slicing using double colons (::).

What do you mean by set and how it is different to lists and tuples?

CHAPTER 5

Dictionaries and Modules

Introduction

In this chapter, we will learn about dictionaries and modules. A dictionary is a commonly used data structure that returns data as key-value pairs. Dictionaries use key-value pairs to store and retrieve data efficiently. Python uses modules to split code into individual files, which makes it easier to organize and reuse. Import statement is used for implementing modules.

Structure

In this chapter, the following topics are covered:

Dictionary

Modules

Objectives

The objective of this chapter is to make learners aware of the fundamentals of dictionaries and modules in Python. At the end of this chapter, learners can write the simple programs of dictionaries and modules. Apart from that, you can also troubleshoot the errors. Learners will become proficient in building and working with dictionaries and modules, gain experience writing basic programs with dictionaries and modules and learn efficient methods for locating and fixing Python code bugs through troubleshooting.

Dictionaries in Python

Dictionaries in Python are a set of key value pairs in an unordered form. The empty dictionary in Python is represented in curly braces {}. Here, each element has a mapping with a certain value. Here, the element represents the index, an integer, or a string, just like associative arrays in PHP. It is a built-in data structure that allows you to store and manage data in a key-value pair format.

Note: Syntax of an empty dictionary is :{}. In Python, the dictionary has no order. Keys and values can be written in single, double, and triple quotes as well because Python supports it, but the output will be in single quotes only.

Example 5.1: Write a program for the dictionary

In this example, an empty dictionary {} is created. In a dictionary, we can add a new key-value pair by simply assigning a value to a new key. Apart from that, we can also modify the value associated with a key by assigning a new value to it.

Code 5.1:

```
course = {} # Syntax of Empty Dictionary

course["Java"]="4 weeks only"

course["DotNet"]="6 weeks only"

course["php"]="6 weeks only"

course["Android"]="8 weeks only"

print(course["Android"])

print(course["php"])
```

The program window of Example 5.1 is shown in Figure

Figure 5.1: Example of dictionary

The output of Example 5.1 is shown in Figure

Figure 5.2: Output using dictionaries

Example 5.2: Accessing values in a dictionary using index and arrays

In this example, a dictionary is created named course, and values are accessed using numerical index and associative arrays.

Code 5.2:

```
course = {}

course["Java"]="Used in Mobile Applications"

course["DotNet"]="For Web Based Applications"

course["1"]="php an open source for web applications"

course["2"]="Android for Mobile Computing"

print(course["1"])

print(course["2"])

print(course["DotNet"])
```

The program window of Example 5.2 is shown in Figure

```
File  Edit  Format  Run  Options  Window  Help
course = {}
course["Java"]="Used in Mobile Applications"
course["DotNet"]="For Web Based Applications"
course["1"]="php an open source for web applications"
course["2"]="Android for Mobile Computing"
print(course["1"])
print(course["2"])
print(course["DotNet"])
```

Figure 5.3: Numerical index and associative array using dictionary

The output of the Example 5.2 is shown in Figure

```
>>>
    = RESTART: C:/Users/DELL/AppData/Local/Programs/Python/
    php an open source for web applications
    Android for Mobile Computing
    For Web Based Applications
```

Figure 5.4: Output of numerical index and associative array using dictionary

Example 5.3: Dictionary example for addition and deletion of key-value pairs

In this example, a complete dictionary is printed as a key-value pair, and a delete function del() is used to delete a key-value pair.

Code 5.3:

```
course = {}

course["Java"]="Used in Mobile Applications"

course["DotNet"]="For Web Based Applications"

course["1"]="php an open source for web applications"

course["2"]="Android for Mobile Computing"

# Value can be added to a numeric key also

course[3]="Python is an interpreter-based"

print(course) #print complete dictionary in key-value pairs form

print(course["1"])

course["Python"] = "Machine Learning"  #add new dictionary key-value pair
```

del course["Java"] # delete this key-value pair

print(course["DotNet"])

print(course[3]) #retrieving the value to a numeric key

The program window of Example 5.3 is shown in Figure

```
course = {}
course["Java"]="Used in Mobile Applications"
course["DotNet"]="For Web Based Applications"
course["1"]="php an open source for web applications"
course["2"]="Android for Mobile Computing"
# Value can be added to a numeric key also
course[3]="Python is an interpreter-based"

print(course)   #print complete dictionary in key-value pairs form
print(course["1"])
course["Python"] = "Machine Learning"   #add new dictionary key-value pair
del course["Java"]      # delete this key-value pair
print(course["DotNet"])
print(course[3])#retrieving the value to a numeric key
```

Figure 5.5: Program window of Example 5.3

The output of the Example 5.3 is shown in Figure

```
{'Java': 'Used in Mobile Applications', 'DotNet': 'For Web Based Applications',
'1': 'php an open source for web applications', '2': 'Android for Mobile Computi
ng', 3: 'Python is an interpreter-based'}
php an open source for web applications
For Web Based Applications
Python is an interpreter-based
```

Figure 5.6: Output window of Example 5.3

Nested dictionary

A nested dictionary in Python is a dictionary that contains another dictionary (or dictionaries) as one of its values.

Example 5.4: Example of nested dictionary

In this example, a nested dictionary is defined inside the outer dictionary one is course1 and the other is Using values of dictionary course1 and course2 are displayed.

Code 5.4:

```
dict1 = {

"course1": {

    "name": "Python Programming",

    "fees": 8000,

    "duration": "8 weeks"

        },

"course2": {

    "name": "Java Programming",

    "fees": 6000,
```

"duration": "6 weeks"

}

}

print("The values are:",dict1['course1']['name'])

print("The values are:",dict1['course2']['fees'])

The program window of Example 5.4 is shown in Figure

```
dict1 = {
"course1": {
    "name": "Python Programming",
    "fees": 8000,
    "duration": "8 weeks"
          },
"course2": {
    "name": "Java Programming",
    "fees": 6000,
    "duration": "6 weeks"
             }
 }

print("The values are:",dict1['course1']['name'])
print("The values are:",dict1['course2']['fees'])
```

Figure 5.7: Nested dictionary

The output window is shown in Figure

```
>>>
= RESTART: C:\Users\LENOVO\AppData\Local
ary.py
The values are: Python Programming
The values are: 6000
```

Figure 5.8: Output of nested dictionary

Example 5.5: Counting the number of digits, lowercase, uppercase, and vowels in a sentence

In this example, an input is given in variable and a cnt variable stores the dictionary value for lowercase, uppercase, digit, and vowel characters. A for loop is iterated over a sentence, and using predefined functions the count is obtained for digits, uppercase, and lowercase letters in a sentence. Apart from it, vowels are defined in a variable and using a for loop, a count of vowels occurring in a sentence is obtained.

Code 5.5:

```
msg = input("Input a sentence:")

cnt = {"digits":0, "lower":0, "upper":0,"vowels":0}

for char in msg:

  if char.isdigit():

      cnt["digits"] = cnt["digits"]+1

  elif char.isupper():

cnt["upper"] = cnt["upper"]+1

  elif char.islower():
```

```
        cnt["lower"] = cnt["lower"]+1

print("No.of Digits are:",cnt['digits'])

print("No.of Uppercase letters are:",cnt['upper'])



print("No.Lowercase letters are:", cnt['lower'])

# Display the output

vowels = 'aeiouAEIOU'

for char in msg:

  if char in vowels:

        cnt["vowels"]=cnt["vowels"]+1

# Display the output

print("No.of Vowels are:", cnt['vowels'])
```

The program window is shown in Figure

```
msg = input("Input a sentence:")
cnt = {"digits":0, "lower":0, "upper":0,"vowels":0}
for char in msg:
    if char.isdigit():
        cnt["digits"] = cnt["digits"]+1
    elif char.isupper():
        cnt["upper"] = cnt["upper"]+1
    elif char.islower():
        cnt["lower"] = cnt["lower"]+1

print("No.of Digits are:",cnt['digits'])
print("No.of Uppercase letters are:",cnt['upper'])
print("No. of Lowercase letters are:", cnt['lower'])
# Display the output
vowels = 'aeiouAEIOU'
for char in msg:
    if char in vowels:
        cnt["vowels"]=cnt["vowels"]+1
# Display the output
print("No.of Vowels are:", cnt['vowels'])
```

Figure 5.9: Counting the number of digits, lowercase, uppercase, and vowels in a sentence

The output window of the example is shown in Figure

```
>>>
    = RESTART: C:/Users/DELL/AppData/Local/Programs
    Input a sentence:PythonProgramming123
    No.of Digits are: 3
    No.of Uppercase letters are: 2
    No.Lowercase letters are: 15
    No.of Vowels are: 4
```

Figure 5.10: Output of Example 5.5

pop() method in a dictionary

A dictionary in Python is a group of key-value pairs. A built-in dictionary method called pop() retrieves the value of the key that is provided and then removes a key-value pair from the dictionary based on that key.

Example 5.6: Example of pop() method in dictionary

In this example, pop() method removes the key-value pair associated with the key java and returns the value of the java key, that is, 2.

Code 5.6:

```
dict1 = {"python": 1,"java":2,"HTML":3}

var1 = dict1.pop("java")

print(var1)

print(dict1)
```

The program window of the example is shown in Figure

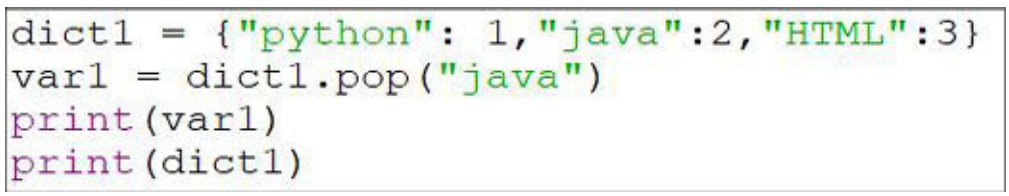


Figure 5.11: pop() method in dictionary

The output window of the example is shown in Figure

```
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/
2
{'python': 1, 'HTML': 3}
...
```

Figure 5.12: Output of pop() method

Modules

A module is a set of statements or just like functions that can be included in an application. Modules can contain variables like arrays, objects, and so on. In modules, we define functions, classes, and variables. It helps in breaking down large program codes into smaller making it easier to work on large projects in collaboration. The advantage of using modules is that we can reuse the code, and it makes the code simpler to understand and maintain. We can create our modules and can import a module into a code using an import statement. With the help of an import statement, we can directly use the in-built mathematical functions like

Example 5.7: Using modules

In this example, a function course() is defined using the def keyword, and this module is saved with the name Then, this module is called inside another Python file,

Firstly, save this module with the filename testmodule.py.

Code 5.7:

```
def course(n):

  print("So you are interested in," + n)
```

The program window in the IDLE is shown in Figure

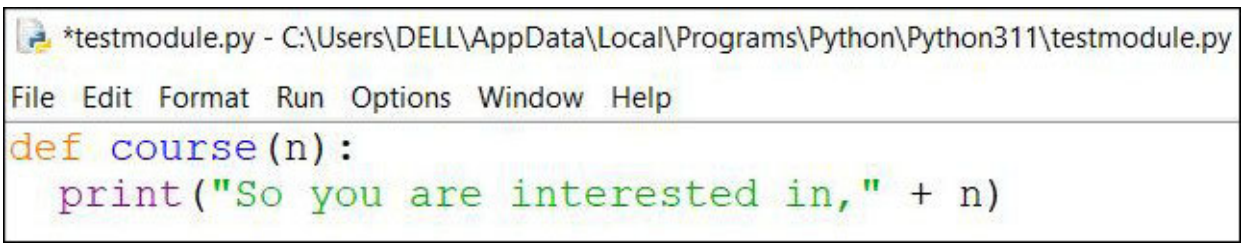

*testmodule.py - C:\Users\DELL\AppData\Local\Programs\Python\Python311\testmodule.py
File Edit Format Run Options Window Help
def course(n):
print("So you are interested in," + n)

Figure 5.13: testmodule.py

Note: The important point to note is this module cannot be directly executed and will work when it is called using an import statement like in the 'testmodule1.py' file. It is for the understanding of the learners that keep both the files 'testmodule.py' and 'testmodule1.py' in the same directory in order

to use the function ‘course’ defined in the ‘testmodule.py’ file in ‘testmodule1.py’ file.

‘testmodule1.py’ is the main script to execute. In this file, testmodule is called using import statement and therefore, you can use the ‘course’ function defined in the ‘testmodule.py’ file.

Code 5.8:

testmodule1.py

```
# import statement is used to import the module

import testmodule

# Here 'python' is passed as a parameter in course function defined in
'testmodule.py'

testmodule.course("Python")
```

The IDLE window for testmodule1.py is shown in Figure

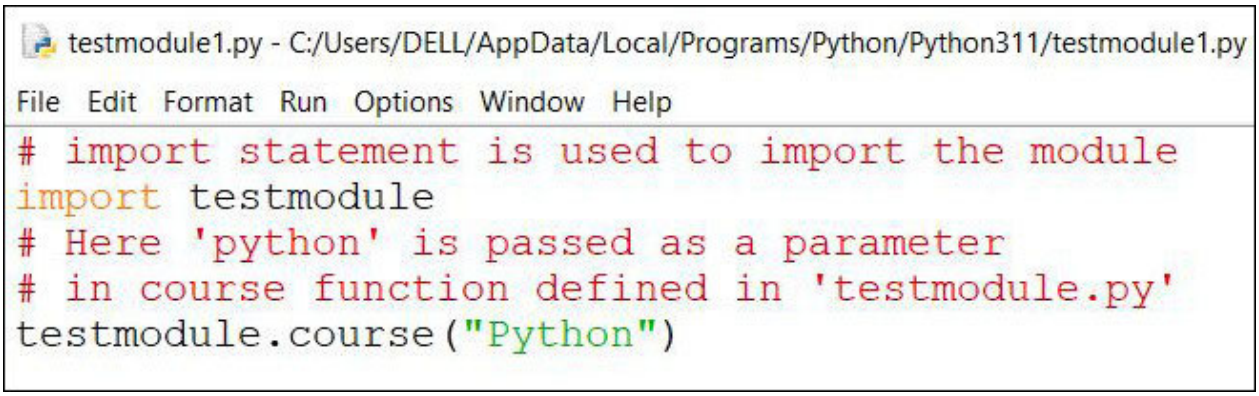

testmodule1.py - C:/Users/DELL/AppData/Local/Programs/Python/Python311/testmodule1.py
File Edit Format Run Options Window Help
# import statement is used to import the module
import testmodule
# Here 'python' is passed as a parameter
# in course function defined in 'testmodule.py'
testmodule.course("Python")

Figure 5.14: testmodule1.py

Note: The “testmodule.course” statement is used in “testmodule1.py” file where testmodule is the module’s name and is saved as a file ‘testmodule.py’. The course is the name of a function defined with the keyword “def” and “Python” is passed as a parameter. An important point to note is that both these python files, that is, ‘testmodule.py’ and ‘testmodule1.py’, are to be saved in the same directory, as module ‘testmodule.py’ is called inside ‘testmodule1.py’.

The output of Example 5.7 is shown in Figure

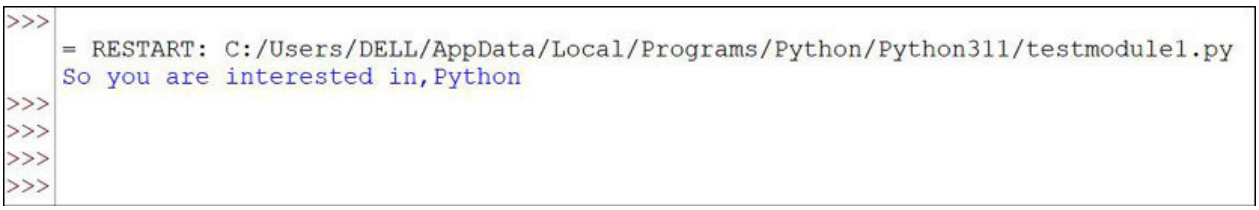


Figure 5.15: Output of testmodule1.py

Example 5.8: Example of using dictionary using modules

In this example, variables of all types are defined in a module named coursemodule.py in the form of a dictionary that is in a key-value pair, and this module is then called in the coursemodule1.py file.

Code 5.9:

coursemodule.py

```
coursename = {

"coursename":"Java",

"fees":4000,

"duration":"Four weeks"

}
```

The IDLE window of coursemodule.py is shown in Figure

Figure 5.16: coursemodule.py

In this program that is, in coursemodule1.py file below, an import statement is used to import the module named coursemodule written above. The dictionary defined in this module, that is, coursename is called in coursemodule1.py and fees and duration are passed as a parameter. The IDLE window of coursemodule1.py file is shown in Figure

Code 5.10:

coursemodule1.py

```
import coursemodule

coursename=coursemodule.coursename["coursename"]

fees = coursemodule.coursename["fees"]

duration = coursemodule.coursename["duration"]

print("The name of course is",  coursename)
```

print("The fees for the course is", fees)

print("The duration for the course is", duration)

The program window looks like the following figure:

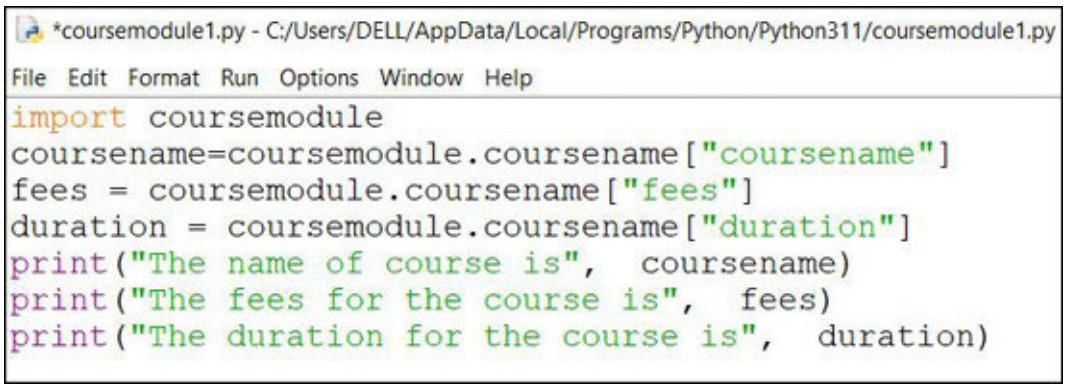

*coursemodule1.py - C:/Users/DELL/AppData/Local/Programs/Python/Python311/coursemodule1.py
File Edit Format Run Options Window Help
import coursemodule
coursename=coursemodule.coursename["coursename"]
fees = coursemodule.coursename["fees"]
duration = coursemodule.coursename["duration"]
print("The name of course is", coursename)
print("The fees for the course is", fees)
print("The duration for the course is", duration)

Figure 5.17: Program window of coursemodule1.py

Note: Ensure that both files (coursemodule.py) and (coursemodule1.py) should be kept in the same directory. When you run the main script that is, coursemodule1.py, it will import the coursemodule.py module and retrieve the values from the coursename dictionary within it.

The output of Example 5.8 is shown in Figure

Figure 5.18: Output window of coursemodule1.py

Example 5.9: Example of importing mathematical functions in a module

In this example, an in-built mathematical function sqrt is called by importing a math module, as shown in the program window in Figure In the sqrt() function, 36 is passed as a parameter, and with the help of the in-built square root sqrt() function inside the math module, the square root of 36 is calculated.

Code 5.11:

```
import math

print('The squareroot of 36 is',math.sqrt(36))
```

Take a look at the following program window:

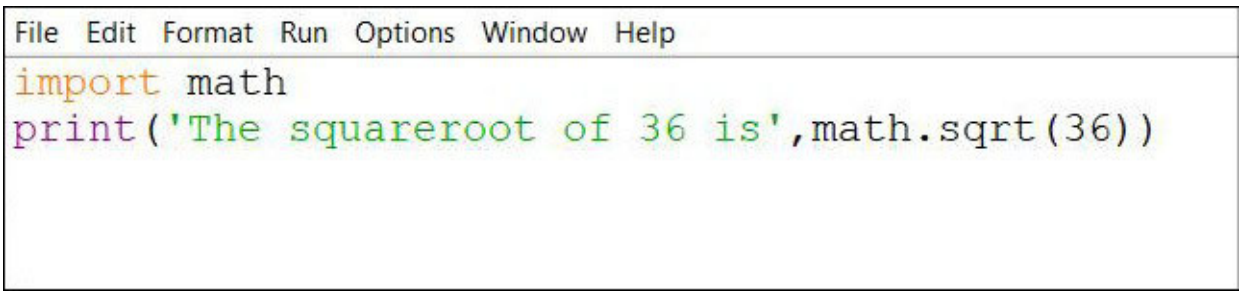


Figure 5.19: Program window of Example 5.9

The output of Example 5.9 is shown in Figure

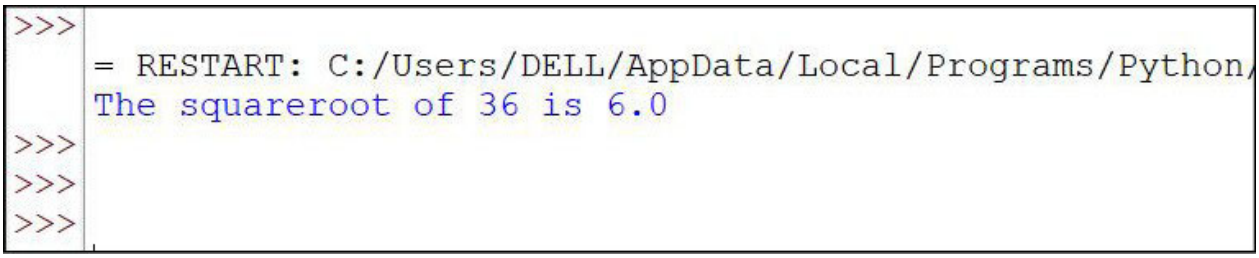


Figure 5.20: Output window of Example 5.9

Note: We can also import specific names, that is, functions, classes from a module instead of importing the whole module as shown below for the above Example 5.9.

Code 5.12:

```
from math import sqrt

print('The squareroot of 36 is', sqrt(36))
```

Conclusion

In this chapter, we have discussed the fundamentals of dictionaries and modules with the help of illustrative examples. In the dictionary, we started with the fundamental concepts and performed operations like adding and deleting data in a dictionary. We also explained the modules with basic examples followed by pre-defined mathematical functions.

Points to remember

Dictionary in Python is a set of key-value pairs in an unordered form and is represented in curly braces {}. Here, each element has a mapping with a certain value.

In dictionaries, key-value pairs contain values of different data types.

A module is a set of statements or just like functions that can be included in an application. Modules can contain variables like arrays, objects, and so on. In modules, we define functions, classes, and variables, it helps in breaking down large program codes into smaller making it easier to work on large projects in association.

pop() is a built-in dictionary method that retrieves the value of the provided key and then removes a key-value pair from the dictionary based on that key.

Exercises

What do you mean by Dictionary? Give a suitable example.

What do you mean by module? Write the advantages of using modules.

Discuss modules with the help of some mathematical functions.

Write a python program to count the frequency of characters in a given string and store the result in a dictionary.

Create a module named math_functions.py containing functions to calculate the area of a circle, rectangle, and triangle. Import this module into your main script and use these functions to calculate the areas of shapes with user-provided dimensions.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 6

File Handling and Databases

Introduction

This chapter will cover the topics related to file handling and databases. File handling in Python enables one to work with files using the built-in functions and libraries. In databases, we will discuss the concept of data definition language and data manipulation language commands for database connections.

Structure

The following topics are covered in this chapter:

Concept of file handling

Database connections in Python

Operations like inserting, updating, reading, and deleting records in Python

Objectives

By the end of this chapter, learners will have a basic understanding of file handling and gain knowledge of data definition language and data manipulation language commands in the context of database management systems. They will also learn the operations of record insertion, reading data, and updating and deleting data in Python. Learners will develop a foundational understanding of file handling in Python, including the concepts of opening, reading, writing, appending, and closing files.

File and modes used in file handling

A file is a stream of data. It is an object that stores data, information, and so on. In Python, the file-handling concepts are the same as in other languages like C language. In Python, there are functions for reading, updating, reading, and deleting files. The following are the modes used in file handling:

r for reading

w for writing

a for appending

r+ for both reading and writing

open() is the built-in function to open a file in Python. open() function takes two parameters. One is filename, and another is mode used. To close a file that has already been opened with the open() method in Python, use the close() function. It is best practice to explicitly close a file you have opened using open() once you have finished using it. This ensures that the file's resources, including file identifiers, are released and that any changes made are flushed to disk.

Example 6.1: Example for reading a file

In this program, myfile.txt is the filename, and r is the read mode for reading. To execute the example, the file myfile.txt should be physically saved in the drive.

Code 6.1:

f = open ('c:\\myfile.txt', 'r')

print (f.read())

f.close()

The program in IDLE is shown in Figure

```
f = open ('c:\\myfile.txt', 'r')
print(f.read())
f.close()
```

Figure 6.1: Reading a data from the file

Note: read() method is used for reading the file's contents.

The output of Example 6.1 is shown in Figure

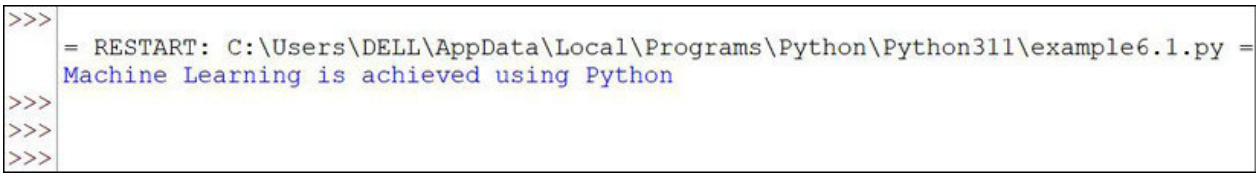


Figure 6.2: Output window of reading a data from the file

Note: The file myfile.txt is to be created first and then save that file in the C drive.

Example 6.2: Example for reading every line of the file

In this example, the program will read the data from a file myfile.txt and print every line of the file one by one.

Code 6.2:

```
f = open ('myfile.txt', 'r')

for each in f:

    print (each)
```

The program window of Example 6.2 in IDLE is shown in Figure

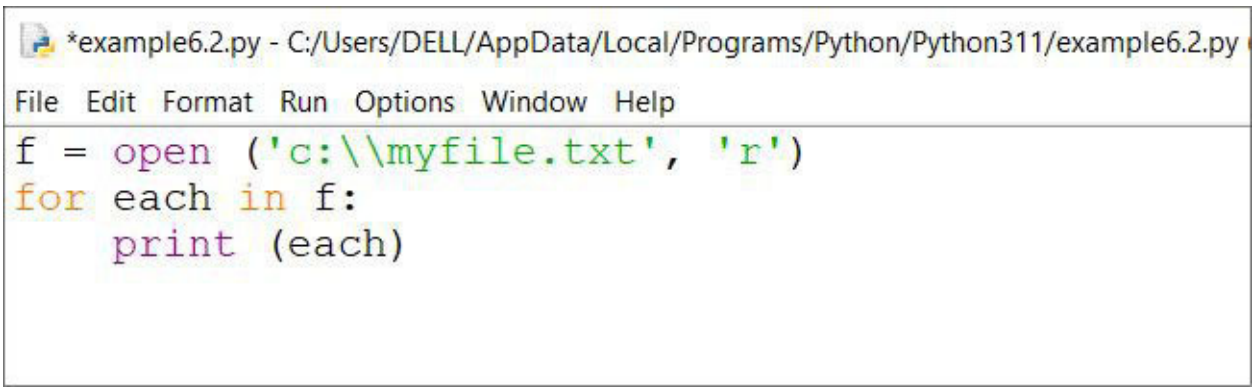

*example6.2.py - C:/Users/DELL/AppData/Local/Programs/Python/Python311/example6.2.py
File Edit Format Run Options Window Help
f = open ('c:\\myfile.txt', 'r')
for each in f:
print (each)

Figure 6.3: Reading data from a file line-by-line

The output of Example 6.2 is shown in Figure

>>>
= RESTART: C:/Users/DELL/AppData/Local/Programs/Python
Machine Learning is achieved using Python.
Python is developed in C Language.

Figure 6.4: Output window of Example 6.2

Example 6.3: Write data into file using w mode

In this example, w mode is used for writing data into a file.

Code 6.3:

```
f = open ('myfile.txt','w')

f.write("W mode is used for writing")

f.close()
```

The IDLE window is shown in Figure

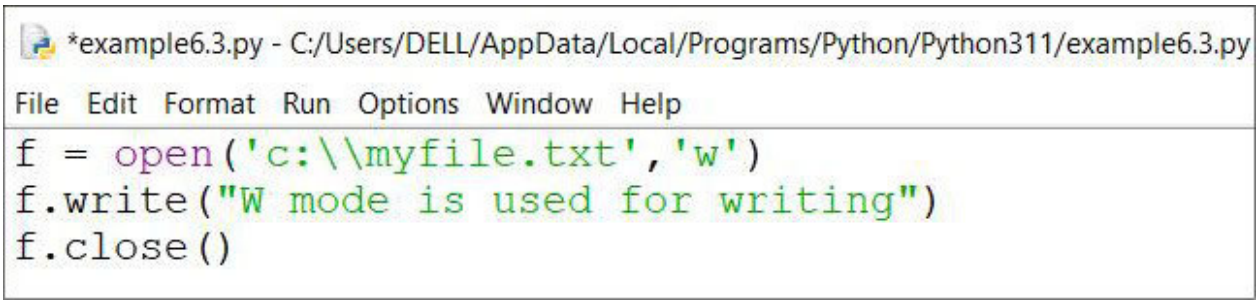

*example6.3.py - C:/Users/DELL/AppData/Local/Programs/Python/Python311/example6.3.py
File Edit Format Run Options Window Help
f = open('c:\\myfile.txt','w')
f.write("W mode is used for writing")
f.close()

Figure 6.5: Writing data into a file using w mode

Using w mode, on opening the file the previous data written in the file will be overwritten with the new data, as shown in Figure

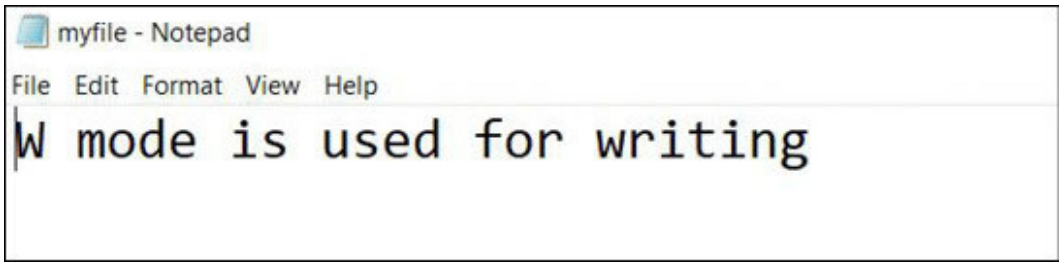


Figure 6.6: Writing data into a file using the w mode

The above example can be written using with statement as shown in Code 6.4:

Code 6.4:

```
with open('c:\\myfile.txt','w') as f:

    f.write("W mode is used for writing")

    f.close()
```

It is an alternative version for more practice for the above example. The with statement is a useful tool in Python for managing and organizing resources, particularly when working with files. Even if an exception arises while the code block is being executed, the with statement makes that the file is opened and closed correctly after using it.

Example 6.4: Write a program to illustrate append mode

In this example, an append mode is used to add data to an existing file data.

Code 6.5:

```
f = open('myfile.txt','a')

f.write("a means append mode which will add data to the already existing data")

f.close()
```

The program window of Example 6.4 is shown in Figure

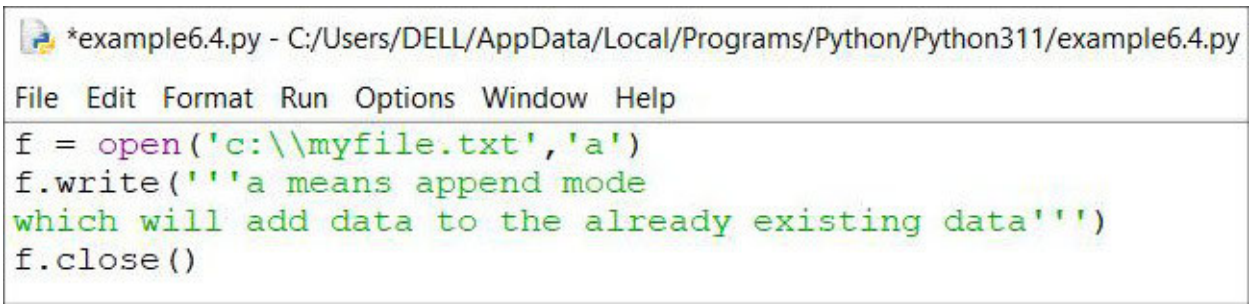

*example6.4.py - C:/Users/DELL/AppData/Local/Programs/Python/Python311/example6.4.py
File Edit Format Run Options Window Help
f = open('c:\\myfile.txt','a')
f.write('''a means append mode
which will add data to the already existing data''')
f.close()

Figure 6.7: Append data into an existing data in a file

Note: The close() command terminates the resources already in use and frees the memory space.

The output of Example 6.4 is shown in Figure where data is added to existing data in a file using append a mode:

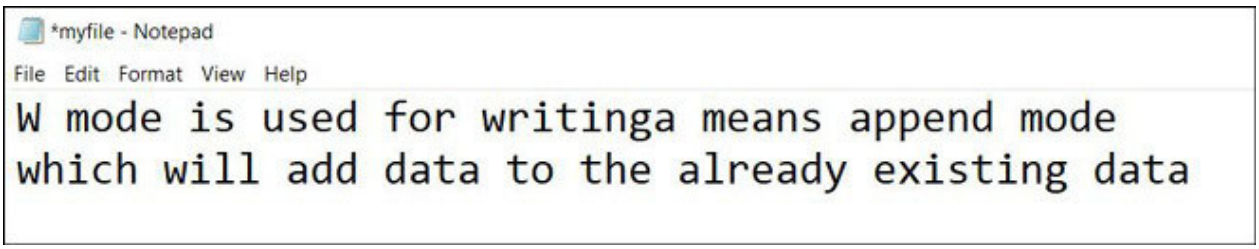

*myfile - Notepad
File Edit Format View Help
W mode is used for writinga means append mode
which will add data to the already existing data

Figure 6.8: Appending data into an existing data in a file

Databases

A database is a structured collection of data that has been arranged, maintained, and stored to facilitate effective information retrieval, modification, and manipulation. Data is kept in a structured way in databases, usually utilizing tables, rows, and columns. Data is arranged in databases as tables that have rows for records and columns for fields. Because each column has a specific data type, data integrity and effective storage are supported. There are various kinds of databases, including relational database management systems which arrange data into tables with well-defined relationships. MySQL, PostgreSQL, Oracle Database, and Microsoft SQL Server are a few examples. NoSQL databases are made to manage data that is unstructured, semi-structured, or changing quickly—for instance, MongoDB.

In this section, Python and database connectivity examples using MySQL are discussed. In the programs, DDL, data query language DML, data control language and transaction control language commands are used.

Data definition language

DDL commands are used to create and modify the database structure. The commonly used DDL commands are: create, drop, alter, and so on. The syntax of the “create table” command is shown below:

```
create table studenttbl(id  int, name varchar(20), address varchar(20));
```

In this command, the table name is “student,” and there are three fields: id, name, and address. The data type of the id field is “int”, the name field is “varchar,” and size is 20, the address field is “varchar”, size is 20.

The drop command is used to delete the object or table from the database. The syntax of the drop command is as follows:

```
drop  table studenttbl;
```

The alter command is used to alter the database structures. The syntax of alter command to add new column “mobno” and datatype integer is as follows:

```
alter  table studenttbl add mobno int;
```

Data query language

The DQL command is used to fetch the records from the database based on the query passed. The “select” statement is used to retrieve the records. The syntax of the select statement for fetching all records and for fetching specific records from the table is as follows:

```
select * from studenttbl;

select * from studenttbl where id = 1;

# where clause is used here to fetch the record having id = 1.
```

Data manipulation language

The DML commands are used to insert, update, and delete the records from the table. The syntax of the insert command is as follows:

```
insert into studenttbl values(1, 'anil', 'delhi');
```

The syntax of the update statement is as follows:

```
update table studenttbl set name = 'anil kumar' where id = 1;
```

In the update query, the where clause is used to update the student's name based on his ID. The syntax of the delete statement is as follows:

```
delete  from studenttbl where id = 1;
```

Data control language

The DCL commands are Grant and Grant command is used to give the users access privileges to the database, whereas Revoke withdraws users' access privileges.

Transaction control language

The commonly used TCL commands are Commit and The Commit command is used to commit a transaction, whereas the Rollback command rollbacks a transaction in case any error occurs.

The syntax and output of the DDL, DQL, and DML commands in the MySQL command terminal, assuming the database name and table name, are shown in Figure 6.9 and Figure

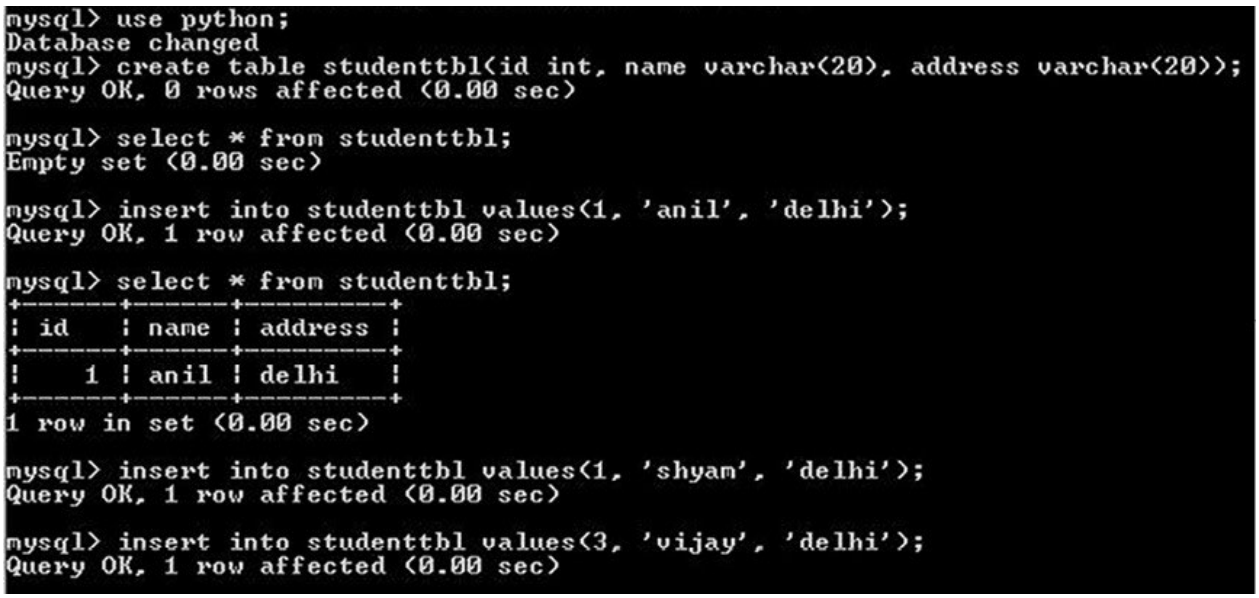

mysql> use python;
Database changed
mysql> create table studenttbl(id int, name varchar(20), address varchar(20));
Query OK, 0 rows affected (0.00 sec)
mysql> select * from studenttbl;
Empty set (0.00 sec)
mysql> insert into studenttbl values(1, 'anil', 'delhi');
Query OK, 1 row affected (0.00 sec)
mysql> select * from studenttbl;
+------+------+---------+
| id | name | address |
+------+------+---------+
| 1 | anil | delhi |
+------+------+---------+
1 row in set (0.00 sec)
mysql> insert into studenttbl values(1, 'shyam', 'delhi');
Query OK, 1 row affected (0.00 sec)
mysql> insert into studenttbl values(3, 'vijay', 'delhi');
Query OK, 1 row affected (0.00 sec)

Figure 6.9: DDL, DQL and DML commands

Take a look at the following figure:

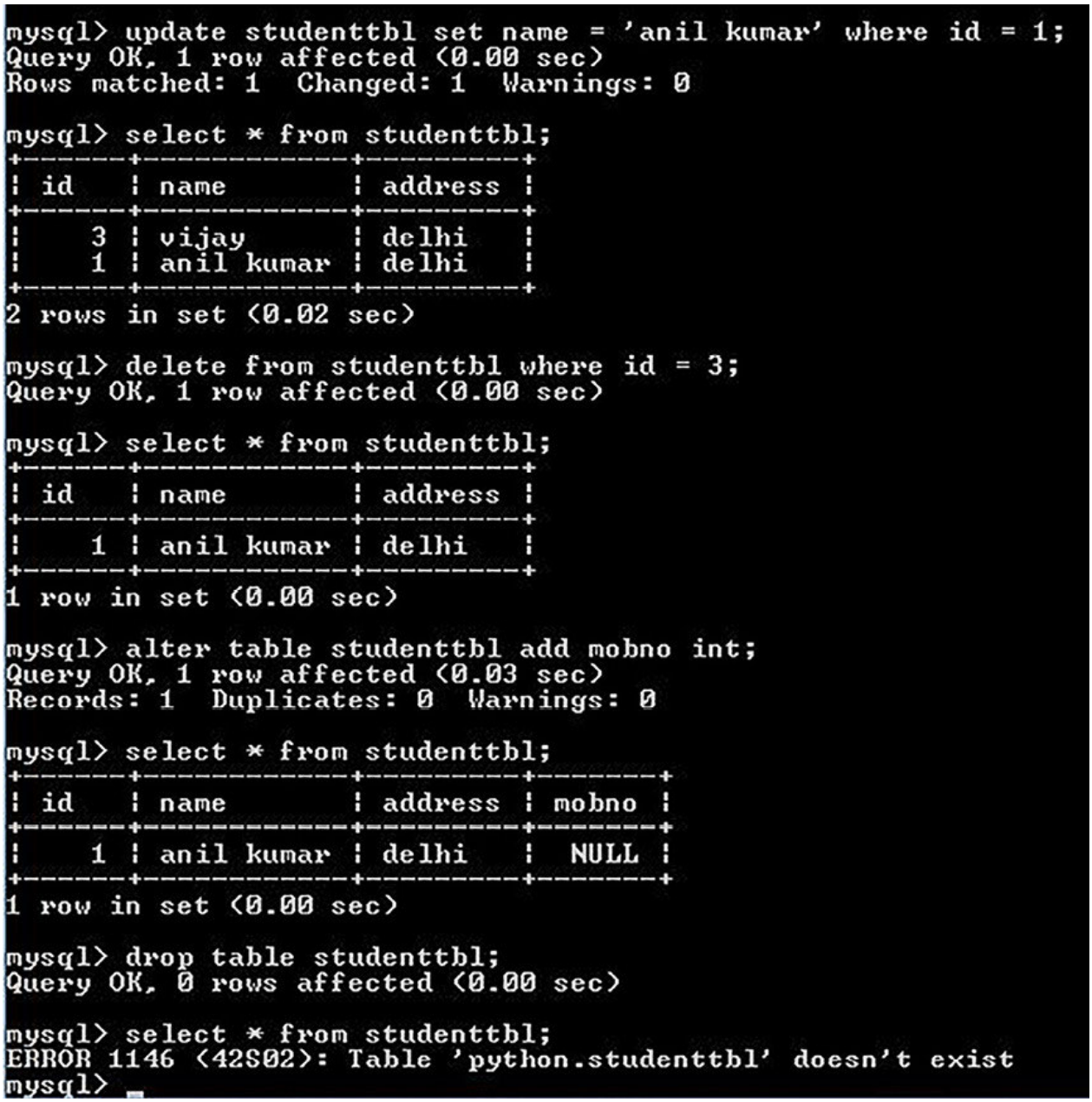

mysql> update studenttbl set name = 'anil kumar' where id = 1;
Query OK, 1 row affected (0.00 sec)
Rows matched: 1 Changed: 1 Warnings: 0
mysql> select * from studenttbl;
+------+------------+---------+
| id | name | address |
+------+------------+---------+
| 3 | vijay | delhi |
| 1 | anil kumar | delhi |
+------+------------+---------+
2 rows in set (0.02 sec)
mysql> delete from studenttbl where id = 3;
Query OK, 1 row affected (0.00 sec)
mysql> select * from studenttbl;
+------+------------+---------+
| id | name | address |
+------+------------+---------+
| 1 | anil kumar | delhi |
+------+------------+---------+
1 row in set (0.00 sec)
mysql> alter table studenttbl add mobno int;
Query OK, 1 row affected (0.03 sec)
Records: 1 Duplicates: 0 Warnings: 0
mysql> select * from studenttbl;
+------+------------+---------+--------+
| id | name | address | mobno |
+------+------------+---------+--------+
| 1 | anil kumar | delhi | NULL |
+------+------------+---------+--------+
1 row in set (0.00 sec)
mysql> drop table studenttbl;
Query OK, 0 rows affected (0.00 sec)
mysql> select * from studenttbl;
ERROR 1146 (42S02): Table 'python.studenttbl' doesn't exist
mysql>

Figure 6.10: DQL and DML commands

In Python, MySQLdb database is used as an interface for connecting to a database server. PostgreSQL and MySQL are two of the most common open-source databases for storing Python web-based application data. PostgreSQL is the relational database for working with Python web applications.

Note: To establish a database connection in Python, we have to install the MySQL Connector package. For that, install the MySQL server and the

MySQL Connector for Python from MySQL Installer, as shown in Figure

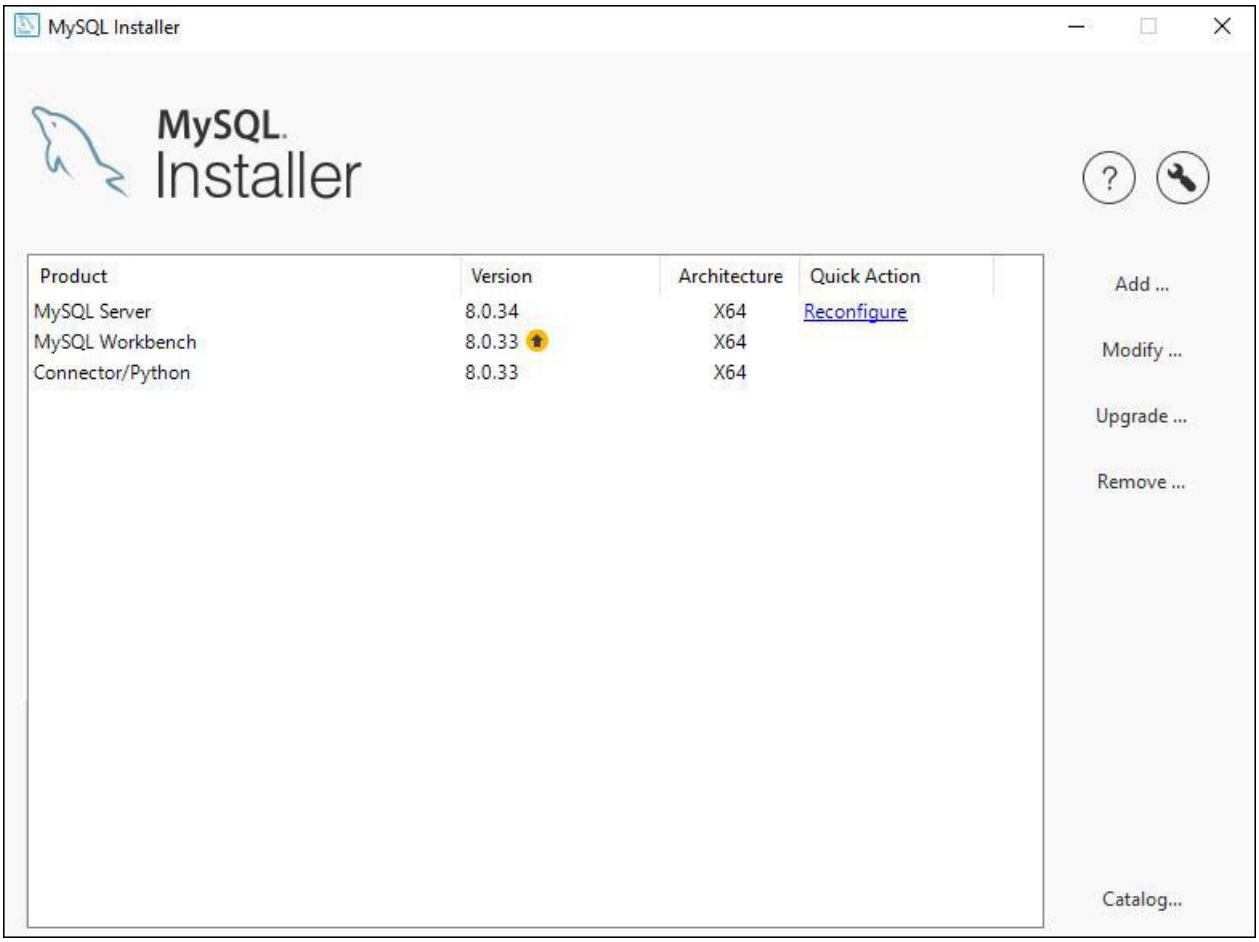


Figure 6.11: MySQL Installer for installing MySQL Server and connector for Python

The window in Figure 6.12 shows the successful installation of the connector for Python. Similarly, install the MySQL server and configure the username, and its password settings. In the examples below, the username is root, and the password is Take a look at the following figure:

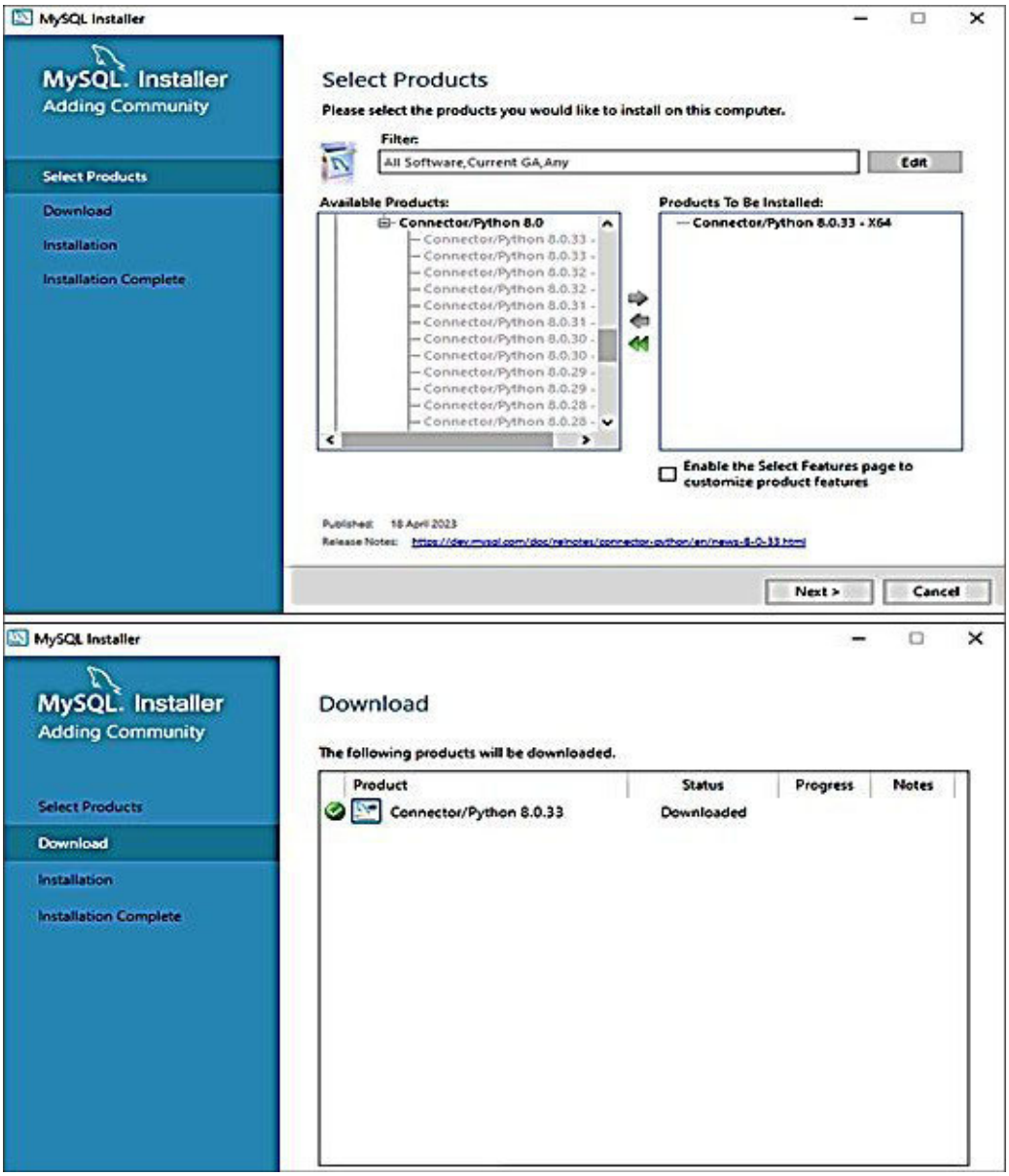


Figure 6.12: Installing the MySQL Connector for Python

Similarly, we need to install the MySQL Server from the MySQL Installer, as shown in Figure

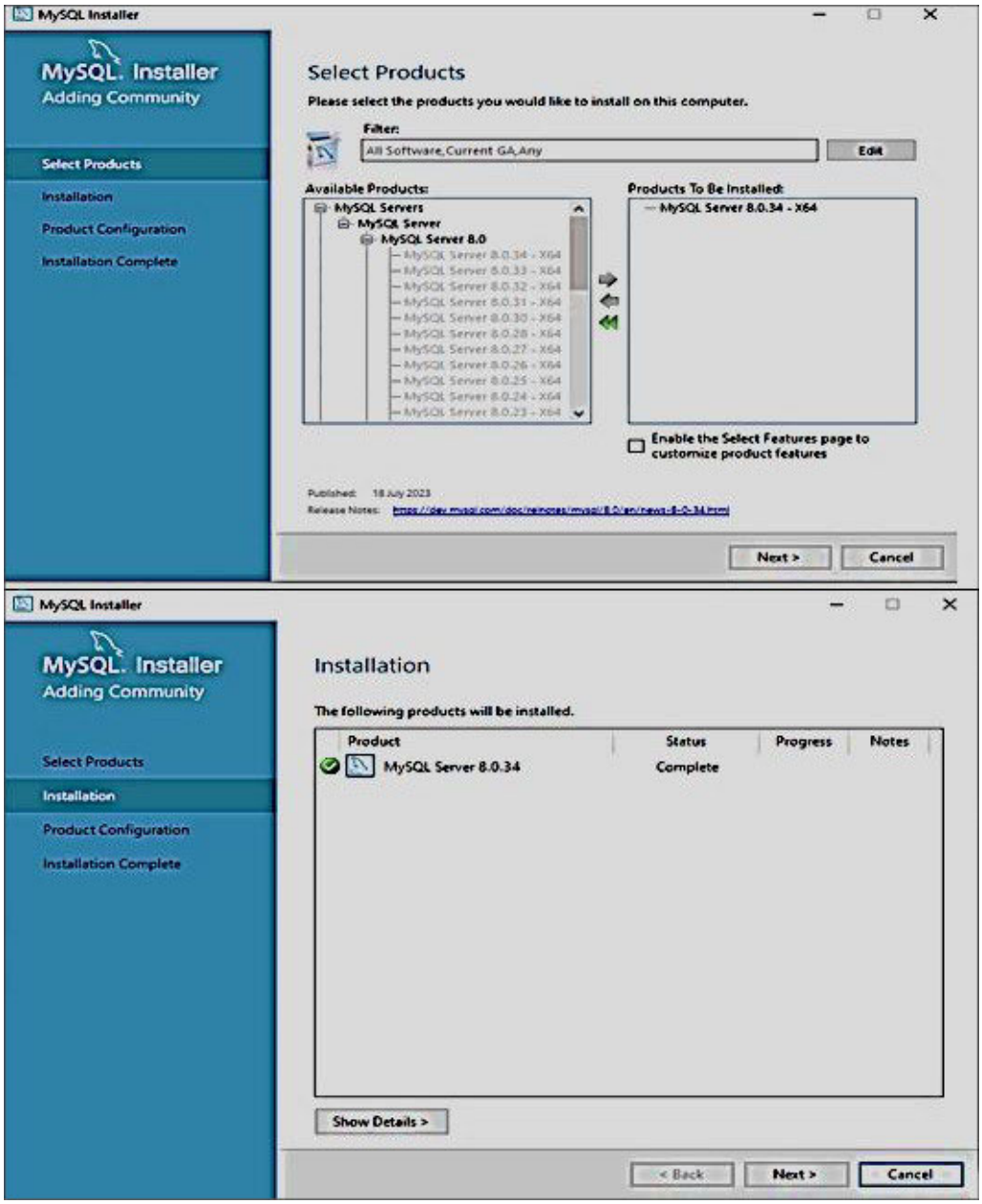


Figure 6.13: Installing the MySQL Server from the MySQL Installer

Now, we need to test whether the connector for Python is installed or not in the IDLE shell, as shown in Figure using the statement import

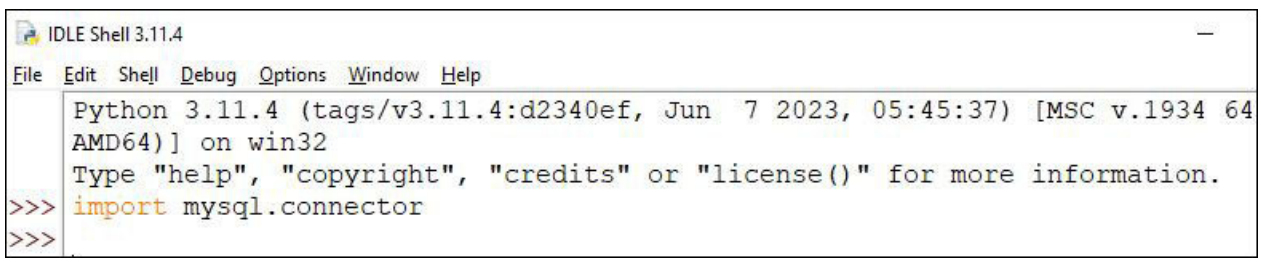

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> import mysql.connector
>>>

Figure 6.14: Testing of MySQL Connector

The MySQL Connector is ready to use as no errors flashed after executing this statement. Now, we must create a connection to your MySQL database by providing the necessary connection details, such as localhost information, username, password, and database name in the program itself.

Example 6.5: Write a program for database and table creation

In this example, MySQL Connector for Python is imported, and the connection is established. A database student is created, and a SQL query is written for creating a table named studenttbl with the attributes and The datatype of all the attributes is also defined in the SQL query. The SQL query for creating a database and table is executed with the help of a cursor statement. The connection is finally committed, and the connection is closed.

Code 6.6:

```
import mysql.connector

# Connect to the MySQL server

conn=mysql.connector.connect(

    host="localhost",

    user="root",

    password="sasml"

)

# Create a database
```

```
dbname = "student"

sqlquery = f"CREATE DATABASE IF NOT EXISTS {dbname};"

cursor=conn.cursor()



cursor.execute(sqlquery)

# Switch to the new database

conn.database=dbname

# Define a SQL command to create a table

TABLE IF NOT EXISTS studenttbl (regid INT AUTO_INCREMENT
PRIMARY KEY,first_name VARCHAR(255) NOT NULL,last_name
VARCHAR(255) NOT NULL,course VARCHAR(255),mobile

conn.commit()

conn.close()

print("Database and table created successfully!")
```

The program in the IDLE window is shown in Figure

```
import mysql.connector
# Connect to the MySQL server
conn=mysql.connector.connect(
    host="localhost",
    user="root",
    password="sasml"
)
# Create a database
dbname = "student"
sqlquery = f"CREATE DATABASE IF NOT EXISTS {dbname};"
cursor=conn.cursor()
cursor.execute(sqlquery)
# Switch to the new database
conn.database=dbname
# Define a SQL command to create a table
cursor.execute("""CREATE TABLE IF NOT EXISTS studenttbl
               (regid INT AUTO_INCREMENT PRIMARY KEY,first_name VARCHAR(255) NOT NULL,
                last_name VARCHAR(255) NOT NULL,course VARCHAR(255),mobile INT)""")
conn.commit()
conn.close()
print("Database and table created successfully!")
```

Figure 6.15: Program to create a database and a table

The output of the example is shown in Figure

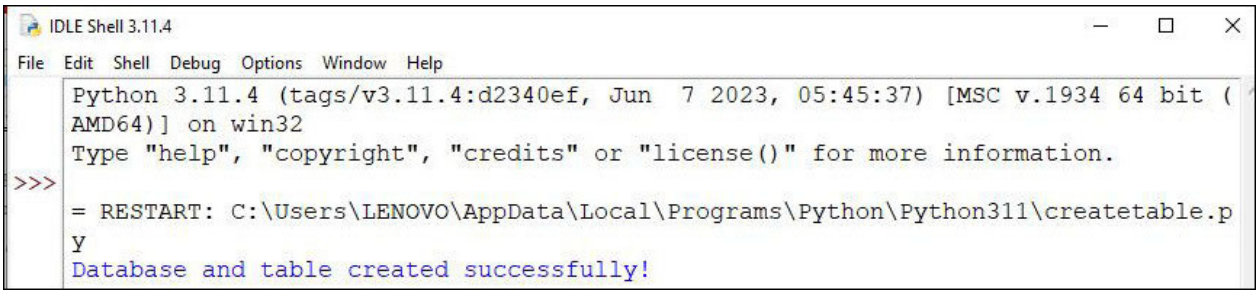

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64 bit (
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: C:\Users\LENOVO\AppData\Local\Programs\Python\Python311\createtable.p
y
Database and table created successfully!

Figure 6.16: Output window of Example 6.5

Example 6.6: Program to insert a record into a table

In this example, a record is inserted into a table, and again, the first step is to make a database connection with a similar configuration as mentioned in the above example. The SQL query to insert records into a table is written in the example, and the same attributes are mentioned as in the previous example. The IDLE window to insert a record into the table is shown in Figure For multiple lines in a SQL query, we use the symbol,’’’ while starting and ending the query.

Code 6.7:

```
import mysql.connector

# MySQL Database Connection Configuration

conn=mysql.connector.connect(

    host="localhost",

    user="root",

    password="sasml",

    database="student"

)
```

```
# Cursor object and Connection interaction

cursor = conn.cursor()

# Executing SQL Query to insert a record into table 'studenttbl'


INTO studenttbl(regid,first_name,last_name,course,mobile)


# Changes are commit to the database

conn.commit()

# Close the cursor and connection

cursor.close()

conn.close()

print("Record inserted successfully.")
```

Take a look at the following figure:

```
import mysql.connector
# MySQL Database Connection Configuration
conn=mysql.connector.connect(
    host="localhost",
    user="root",
    password="sasml",
    database="student"
)
# Cursor object and Connection interaction
cursor = conn.cursor()
# Executing SQL Query to insert a record into table 'studenttbl'
cursor.execute("""INSERT INTO studenttbl(regid,first_name,last_name,course,mobile)
   VALUES(31,'Anil','Kumar','Python',1234567890)""")
# Changes are commit to the database
conn.commit()
# Close the cursor and connection
cursor.close()
conn.close()
print("Record inserted successfully.")
```

Figure 6.17: Program window to insert a record into a table

The output of the example for successfully inserting a record into a table is shown in Figure

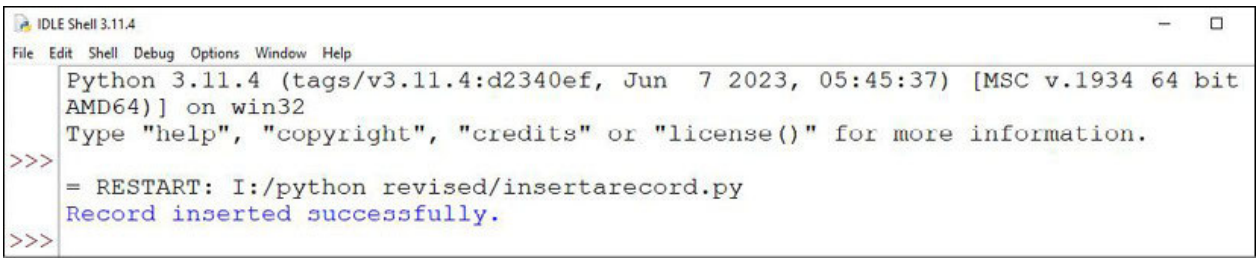

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.1934 64 bit AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>>
= RESTART: I:/python revised/insertarecord.py
Record inserted successfully.
>>>

Figure 6.18: Output window of inserting a record into a table

Example 6.7: To read records from the table and display them

In this example, inserted records are fetched from the table. A database connection is established, and records are read from a table using a SELECT SQL query. To fetch all the records from a table, fetchall() method is used, and using for loop, row-wise records are fetched using indexing, and to display records into the next line, the \n operator is used. A database cursor traverses over the records in a database. A cursor object is used to fetch the records from the database using for loop to traverse the rows for records in a table.

Code 6.8:

```
import mysql.connector

# MySQL Database Connection Configuration

conn=mysql.connector.connect(

host="localhost",

    user="root",

    password="sasml",

    database="student"
```

```
)

# Cursor object and Connection interaction

cursor = conn.cursor()

# Executing SQL Query to insert a record into table 'studenttbl'



cursor.execute("""SELECT * from studenttbl""")

# fetch all the records in a table using fetchall()

display = cursor.fetchall()

# fetch the records using loops and display using '\n'

for x in display:

    print(x,'\n')

# Changes are commit to the database

conn.commit()

# Close the cursor and connection

cursor.close()
```

conn.close()

The program window using IDLE is shown in Figure

```
import mysql.connector
# MySQL Database Connection Configuration
conn=mysql.connector.connect(
    host="localhost",
    user="root",
    password="sasml",
    database="student"
)
# Cursor object and Connection interaction
cursor = conn.cursor()
# Executing SQL Query to insert a record into table 'studenttbl'
cursor.execute("""SELECT * from studenttbl""")
# fetch all the records in a table using fetchall()
display = cursor.fetchall()
# fetch the records using loops and display using '\n'
for x in display:
    print(x,'\n')
# Changes are commit to the database
conn.commit()
# Close the cursor and connection
cursor.close()
conn.close()
```

Figure 6.19: Reading records from a table

The output window is shown in Figure

```
IDLE Shell 3.11.5
File  Edit  Shell  Debug  Options  Window  Help
    Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34) [
    AMD64)] on win32
    Type "help", "copyright", "credits" or "license()" for more i
>>>
    = RESTART: I:\python revised\pythonprograms\example6.7.py
    (32, 'Anil', 'Kumar', 'Python', 1234567890)
```

Figure 6.20: Output window of displaying records

Example 6.8: Write a program to update the data

In this program, an update query is used to update the record using the where clause. Here, and mobile are the attribute names. The table name is studenttbl and the database name is

Code 6.9:

```
import mysql.connector

# MySQL Database Connection Configuration

conn=mysql.connector.connect(

    host="localhost",

    user="root",

    password="sasml",

    database="student"

)

# Cursor object and Connection interaction

cursor = conn.cursor()
```

```
# Executing SQL Query to insert a record into table 'studenttbl'

studenttbl SET first_name =




            where regid =

# fetch all the records in a table using fetchall()

display = cursor.fetchall()

conn.commit()

* FROM studenttbl

            where regid =

display = cursor.fetchall()

# fetch the records using loops and display using '\n'

for x in display:

    print(x,'\n')

# Changes are commit to the database
```

conn.commit()

Close the cursor and connection

cursor.close()

conn.close()

The program window of Example 6.8 is shown in Figure

```
import mysql.connector
# MySQL Database Connection Configuration
conn=mysql.connector.connect(
    host="localhost",
    user="root",
    password="sasml",
    database="student"
)
# Cursor object and Connection interaction
cursor = conn.cursor()
# Executing SQL Query to insert a record into table 'studenttbl'
cursor.execute("""UPDATE studenttbl SET first_name = 'Ajay',
                last_name='Singh',course='Java',mobile='1234567890'
                where regid = 32""")
# fetch all the records in a table using fetchall()
display = cursor.fetchall()
conn.commit()
cursor.execute("""SELECT * FROM studenttbl
                where regid = 32""")
display = cursor.fetchall()
# fetch the records using loops and display using '\n'
for x in display:
    print(x,'\n')
# Changes are commit to the database
conn.commit()
# Close the cursor and connection
cursor.close()
conn.close()
```

Figure 6.21: Updating a record on the basis of regid

The output of Example 6.8 is shown in Figure

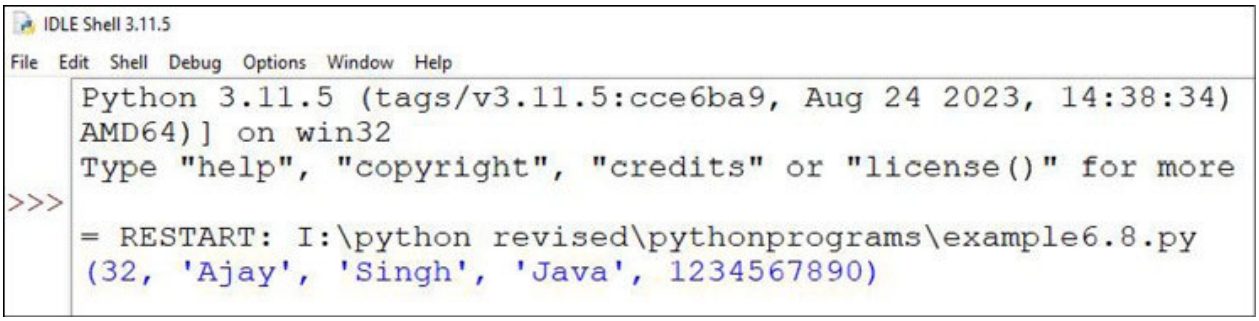

IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34)
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
>>>
= RESTART: I:\python revised\pythonprograms\example6.8.py
(32, 'Ajay', 'Singh', 'Java', 1234567890)

Figure 6.22: Updated the record of a student where regid = ‘32’

Example 6.9: Write a program to delete the data

In this example, a record is deleted from the table using the SQL delete query using the WHERE clause, and the remaining records are displayed through the SELECT SQL query.

Code 6.10:

```
import mysql.connector

# MySQL Database Connection Configuration

conn=mysql.connector.connect(

    host="localhost",

    user="root",

    password="sasml",

    database="student"

)

# Cursor object and Connection interaction

cursor = conn.cursor()
```

```
# Executing SQL Query to insert a record into table 'studenttbl'

FROM studenttbl

                where regid =

# fetch all the records in a table using fetchall()



display = cursor.fetchall()

conn.commit()

cursor.execute("""SELECT * FROM studenttbl""")

display = cursor.fetchall()

# fetch the records using loops and display using '\n'

for x in display:

    print(x,'\n')

# Changes are commit to the database

conn.commit()

# Close the cursor and connection
```

cursor.close()

conn.close()

The IDLE window of Example 6.9 is shown in Figure

```
import mysql.connector
# MySQL Database Connection Configuration
conn=mysql.connector.connect(
    host="localhost",
    user="root",
    password="sasml",
    database="student"
)
# Cursor object and Connection interaction
cursor = conn.cursor()
# Executing SQL Query to insert a record into table 'studenttbl'
cursor.execute("""DELETE FROM studenttbl
                where regid = 32""")
# fetch all the records in a table using fetchall()
display = cursor.fetchall()
conn.commit()
cursor.execute("""SELECT * FROM studenttbl""")
display = cursor.fetchall()
# fetch the records using loops and display using '\n'
for x in display:
    print(x,'\n')
# Changes are commit to the database
conn.commit()
# Close the cursor and connection
cursor.close()
conn.close()
```

Figure 6.23: Deleting a record from the table for regid = ‘32’

The output of Example 6.9 is shown in Figure

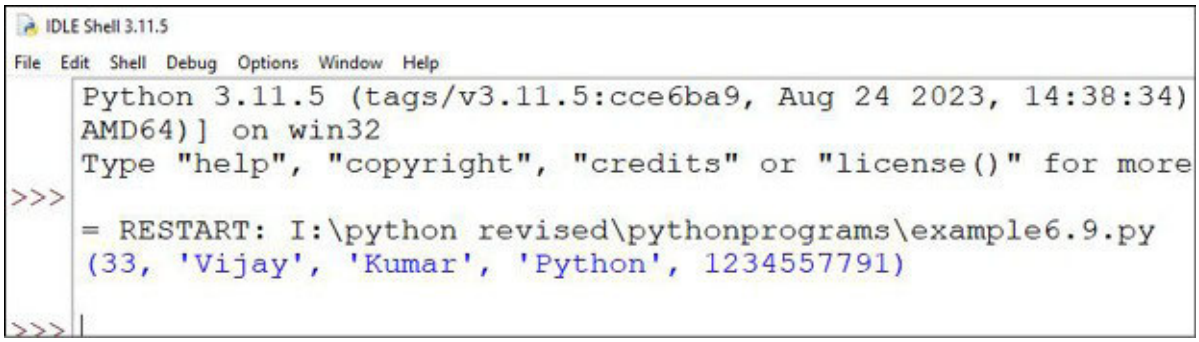

IDLE Shell 3.11.5
File Edit Shell Debug Options Window Help
Python 3.11.5 (tags/v3.11.5:cce6ba9, Aug 24 2023, 14:38:34)
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
>>>
= RESTART: I:\python revised\pythonprograms\example6.9.py
(33, 'Vijay', 'Kumar', 'Python', 1234557791)
>>>

Figure 6.24: Deleted a record where regid=’32’ and displayed other records from a table

Conclusion

In this chapter, we learned the concepts of file handling and modes of operations on file. DDL and DML commands are elaborated with the help of examples. Apart from that, database operations like inserting, updating, reading, and deleting records in Python are explained with examples. In the next chapter, learners will learn about the object-oriented programming features in Python.

Points to remember

To work with a file, we need to open it using the open() function.

There are file modes like r for read, w for write, a for append, and rb for binary files.

To read the content of a file, we use the methods like

For writing data to a file, we can use the write() method and the w mode is used, and for appending data to an existing file, an append mode a is used.

DDL means data definition language, and DML is data manipulation language.

DDL and DML use structured query language queries for managing relational databases.

DDL command examples are: CREATE ALTER DROP TABLE and so on., whereas DML commands are and DELETE and so on.

Exercises

Discuss the file-handling operations with the help of examples.

Define the terms DDL, DML, and DCL. Give examples of each.

Write the steps of database connection in Python.

Write the steps of inserting a record into the table in Python.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 7

Object-Oriented Programming

Introduction

Python supports the object-oriented programming concepts. This chapter will cover the OOP concepts like classes, objects, inheritance, overriding, and so on. A class is a template for creating objects and is a collection of data members, and methods. An object is an instance of a class. In classes, methods are defined within a class that can perform actions on the object using dot notation.

Structure

In this chapter, the following topics are covered:

Class and objects

Inheritance

Overriding

Encapsulation

Objectives

In this chapter, readers will learn the fundamentals of class and objects, along with other features of OOPs, including inheritance, overriding, and encapsulation, illustrated with the help of suitable examples.

Classes and objects

Class and objects are the basic building blocks of OOP in Python. A programming paradigm, known as object-oriented uses objects to organize and structure code. A class serves as a template or model for building objects. A class defines a set of attributes and methods. An object is an instance of a class. These methods and variables are accessed by the object created by a class. Methods are defined inside a class, and the object accesses the functions.

Python has a built-in __init__() function in classes, called when an object is created from a class. The function ___init___() is used to initialize objects, attributes, assign values and is always executed when the object is being initiated. This function assigns values to object properties or performs operations when the object is created.

A basic example of class and objects is covered in Example A class object is created, and it has access to the class's attributes. In this case, a class's constructor, or special method, is defined to initialize an object's attributes upon creation.

Example 7.1: Writing a program for class and objects

In this example, a class is created, and an __init__() function is defined inside a class. Here, x is an object of a class and the value 10 is passed inside the __init__() function. self is a parameter used to access the value using a variable. OurAtt is a variable in which the value 10 is passed.

Code 7.1:

```
class OurClass:

    def __init__(self, a):

        self.OurAtt = a

x = OurClass(10)

print(x.OurAtt)
```

The program window of Example 7.1 is shown in Figure

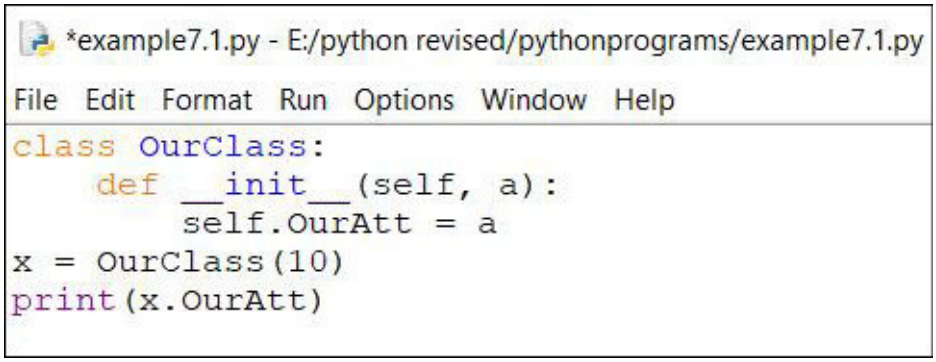


Figure 7.1: Program window for class and objects

The output of this example is shown in Figure

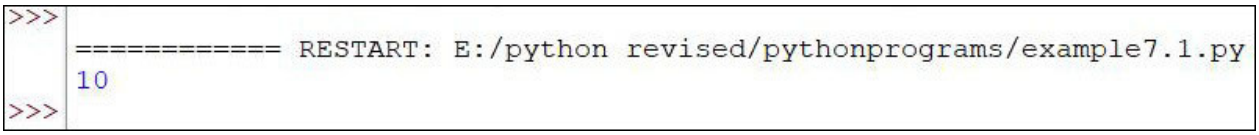


Figure 7.2: Output window of class and objects

Note: The self parameter is used as a first parameter of the function in the class. It acts as a reference to the class itself and is used to access a variable that belongs to the class. It is a class instance that is currently being used and will always point to the current object.

Example 7.2: Writing a program for functions in a class

In this program, Employee is the classname. The name of the defined function is displayCount() and Here, emp2 are the object names and moment the object is created, the values are passed inside the __init__() function. displayEmployee() is called by object A function is defined using the keyword

Code 7.2:

```
class Employee:

  #Employee class is a Base or parent class

  #empCount variable value is initialized to zero and count value will be
#incremented

  empCount = 0

  def __init__(self, name, salary):

      self.name = name

      self.salary = salary

Employee.empCount += 1
```

```
# can also be written as

# Employee.empCount = Employee.empCount + 1

  def displayEmployee(self): # function is defined here

      print("Name :", self.name,  ", Salary:", self.salary)



#emp1 is the first object of Employee class

emp1 = Employee("Akhil", 2000)

#emp2 is the second object of Employee class

emp2 = Employee("Suresh", 5000)

emp1.displayEmployee()

emp2.displayEmployee()

print("Total Employee %d" % Employee.empCount)
```

The program window of Example 7.2 is shown in Figure

```
File  Edit  Format  Run  Options  Window  Help
class Employee:
   'Base class'
   empCount = 0

   def __init__(self, name, salary):
      self.name = name
      self.salary = salary
      Employee.empCount += 1
# can also be written as Employee.empCount = Employee.empCount + 1

   def displayEmployee(self):   # function is defined here
      print("Name:",self.name,",Salary:",self.salary)

"emp1 is the first object of Employee class"
emp1 = Employee("Akhil",2000)
"emp2 is the second object of Employee class"
emp2 = Employee("Suresh",5000)
emp1.displayEmployee()
emp2.displayEmployee()
print("Total Employee %d" % Employee.empCount)
```

Figure 7.3: Program window for using functions in a class

The output of Example 7.2 is shown in Figure

```
>>>
============ RESTART: E:/python revised/pythonprograms/example7.2.py =====
Name: Akhil ,Salary: 2000
Name: Suresh ,Salary: 5000
Total Employee 2
```

Figure 7.4: Output of using functions in a class

Inheritance

One of the core ideas of object-oriented programming is inheritance. It permits a base class or superclass's methods and properties to be inherited by a derived or subclass. This facilitates the creation of a class hierarchy and code reuse. The detail class in Example 7.3 inherits the base class

Example 7.3: Writing a program for inheritance

In this program, class Course is the base class or parent class or super class and class detail is the inherited class inheriting the properties of the base class. Here, x is the object of the Course class, and y is the object of the detail class. When the object x is created, the values are passed inside the __init()__ function, and similar is the case with the y object. The functions Name() and GetDetail() are called with the help of objects x and

Code 7.3:

```
class Course:

    def __init__(self, course, duration):

        self.cname = course

        self.cduration = duration

    def Name(self):

        return self.cname + " " + self.cduration

class detail(Course):

    def __init__(self, course, duration, fees):
```

```
        Course.__init__(self,course, duration)

        self.cfees = fees

    def GetDetail(self):

        return self.Name() + ", " +  self.cfees


x = Course("C++", "4 weeks")

y = detail("PHP", "4 weeks", "4000")

print(x.Name())

print(y.GetDetail())
```

The program window of Example 7.3 is shown in Figure

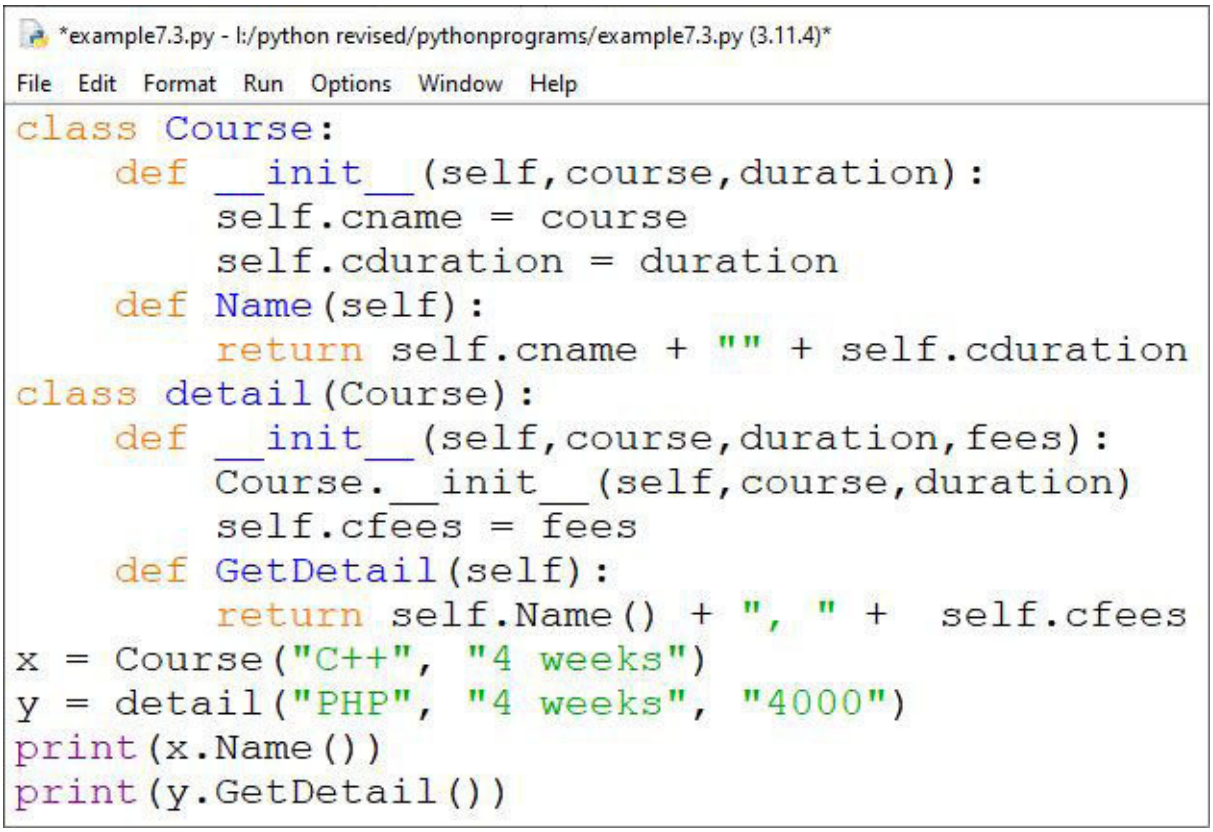


Figure 7.5: Program window of inheritance

The output of Example 7.3 is shown in Figure

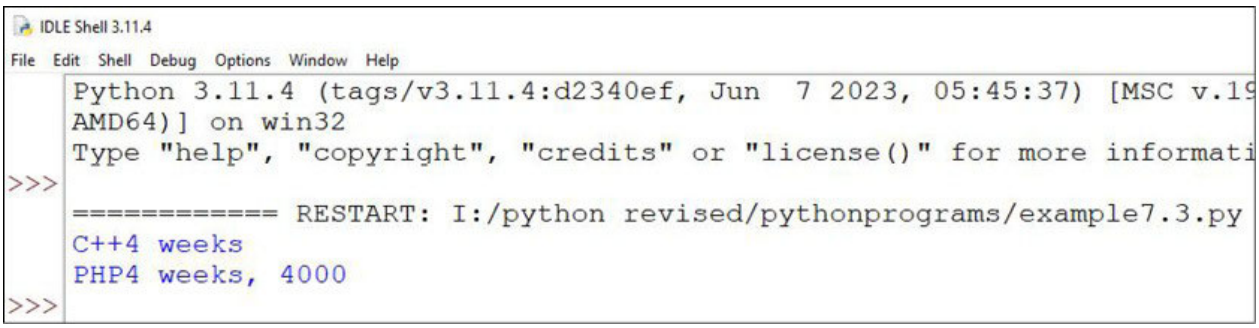

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.19
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more informati
>>>
========== RESTART: I:/python revised/pythonprograms/example7.3.py
C++4 weeks
PHP4 weeks, 4000
>>>

Figure 7.6: Output window of Example 7.3

Overriding

Override means having two methods with the same name performing different tasks. It means that one of the methods overrides the other. If there is any method in the superclass and subclass with the same name, then by executing the method, the method of the corresponding class will be executed. Since the method from the corresponding class came into action, one overrode the other.

The ability of a subclass to give its own implementation for a method that is already defined in its superclass is known as method overriding. A subclass is considered to override a method when it defines one that is already defined in its superclass. The subclass in Example 7.4 that inherits from the base class, is called Since the method getTotal() is defined in both the base class and the subclass, the subclass fees will override the base class method with its own implementation.

Example 7.4: Writing a program for overriding

In this example, two classes are defined: course and fees. In the course class, __init()__ method is defined with three parameters: self, duration, and fees. The values will be passed to these parameters when object of the respective classes are created. The fees class is inheriting the course class. Another method defined in both classes is The object of fees and course class are represented by f and respectively.

Code 7.4:

```
class course():

    def __init__(self,duration,fees):

        self.duration = duration

        self.fees = fees

    def getTotal(self):

        print(self.duration, "is the duration and",self.fees, "is the fees of course")

class fees(course):

    def __init__(self,name):
```

```
        self.name = name

    def getTotal(self):

        print (self.name, "is the name of course")


f = fees("java")

c = course("4 weeks",4000)

f.getTotal()

c.getTotal()
```

The program window of Example 7.4 is shown in Figure

```
class course():
        def __init__(self,duration,fees):
                self.duration = duration
                self.fees = fees
        def getTotal(self):
                print(self.duration,"is the duration and",self.fees,"is the fees of course")
class fees(course):
        def __init__(self,name):
                self.name = name
        def getTotal(self):
                print(self.name," is the name of course")
f = fees("java")
c = course("4 weeks",4000)
f.getTotal()
c.getTotal()
```

Figure 7.7: Program window of overriding

The getTotal() function defined in the course class will print 4 weeks is the duration and 4000 is the fees of course in the example above when it is executed on the object of the course class. In contrast, the f.getTotal()

function executed on the object of the fees class, that is will print java is the name of course from the getTotal() function defined in the fees class.

The output of Example 7.4 is shown in Figure

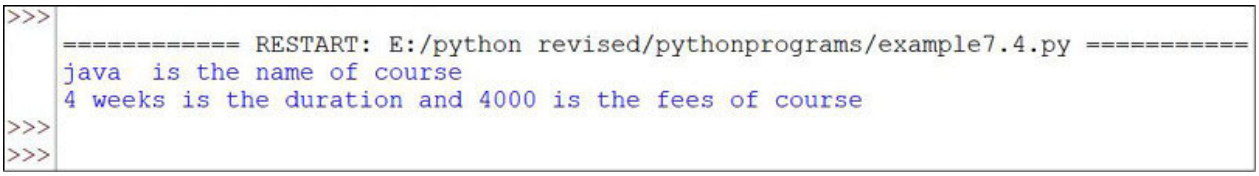

>>>
========== RESTART: E:/python revised/pythonprograms/example7.4.py ==========
java is the name of course
4 weeks is the duration and 4000 is the fees of course
>>>
>>>

Figure 7.8: Output of overriding

Example 7.5: Write a program for inheritance and function overriding

In this example, a class course() is defined and class fees() is inheriting the course() class. Here, getdata() is the name of methods defined in both the classes. The name of both the methods is similar in classes and c is the name of object created of the fees class. The object c will call the method getdata() of fees class. In the fees() subclass, the getdata() method will add the value 5000 to 2000, and the output will result in a 7000 value. In this method, the value 5000 is inherited from the course() parent class.

Code 7.5:

```
class course():

  def __init__(self):

      self.value = 5000

  def getdata(self):

      print (self.value, "is the fees of course")

class fees(course):

# getdata() function is defined here

  def getdata(self):
```

self.value += 2000

print (self.value, "is the increased fees of course")

c is the object of the fees class

c = fees()

getdata() is the function called by the c object

c.getdata()

The program window of Example 7.5 is shown in Figure

```
class course():
  def __init__(self):
      self.value = 5000
  def getdata(self):
      print(self.value, "is the fees of course")

class fees(course):
# getdata() function is defined here
  def getdata(self):
      self.value += 2000
      print(self.value, "is the increased fees of course")

# c is the object of the fees class
c = fees()
# getdata() is the function called by the c object
c.getdata()
```

Figure 7.9: Program window of inheritance and function overriding

The output of Example 7.5 is shown in Figure

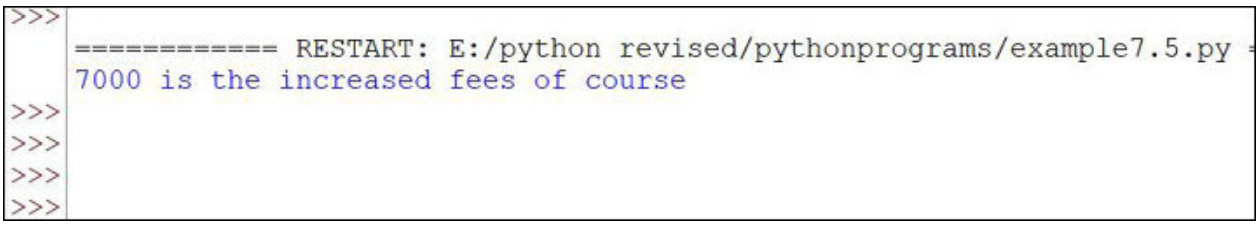


Figure 7.10: Output window of inheritance and function overriding

Encapsulation

Encapsulation is one of the fundamental concepts in OOP. It wraps the variables and methods of a class, providing security of the data. It puts restrictions on the direct accessing of the data using access specifiers, that is, public, private and protected. In case of private modifier, the object cannot access the data members of a class. The protected members of the class are members that cannot be accessed outside but from within the class and its one derived class.

Note: In Python, when utilizing a protected access specifier, attributes and methods meant for internal use within a class or its subclasses are prefixed with a single underscore (‘_’). On the other hand, when utilizing a private access specifier, attributes and methods that are deemed private and not meant to be used directly from outside the class are prefixed with double underscores (‘__’).

Example 7.6: Write a program for encapsulation restricting access to the private variables

In this example, a class Course is defined, and two variables: fees and are created as private variables by prefixing the The values are passed to variables in a constructor and a function getinfo() is defined. An object obj1 of a class is created, and with the help of object the function getinfo() is called to display the values passed to these private variables inside a function. Then, these variable values of course_name are changed using object However, since these variables are private variables, their values cannot be changed. After calling the function getinfo() again, the values remain the same, and the output comes the same.

Code 7.6:

```
class Course:

    __fees=0

    __course_name=""

    def __init__(self):

        self.__fees = 3000

        self.__course_name="Java"

    def getinfo(self):
```

```
        print('course fees is::' + str(self.__fees))


        print('course name is::' + str(self.__course_name))

#obj1 is the object of a class Course

obj1 = Course()

#function getinfo() is called using obj1

obj1.getinfo()

# These are private variables,therefore, cann't change it.

obj1.__fees=4000

obj1.__course_name="C++"

obj1.getinfo()
```

The program window is shown in Figure

```
class Course:
    __fees=0
    __course_name=""
    def __init__(self):
        self.__fees = 3000
        self.__course_name="Java"
    def getinfo(self):
        print('course fees is::' + str(self.__fees))
        print('course name is::' + str(self.__course_name))

#obj1 is the object of a class Course
obj1 = Course()
#function getinfo() is called using obj1
obj1.getinfo()
# These are private variables,therefore, cann't change it.
obj1.__fees=4000
obj1.__course_name="C++"
obj1.getinfo()
```

Figure 7.11: Restricting access to private variables using encapsulation

The output of Example 7.6 is shown in Figure

```
course fees is::3000
course name is::Java
course fees is::3000
course name is::Java
>>>
```

Figure 7.12: Output of Example 7.6

Example 7.7: Write the aforementioned example using public access specifier

The variables fees and course_name restricted in Example 7.6 are now changed to public variables. Their values can now be changed using the object of a class. When a function getinfo() is called the first time, the values displayed are (3000 and Java), but on calling the function getinfo() the second time, the values displayed are (4000 and C++).

Code 7.7:

```
class Course:

    fees=0

    course_name=""

    def __init__(self):

        self.fees = 3000

        self.course_name="Java"

    def getinfo(self):

print('course fees is::' + str(self.fees))

        print('course name is::' + str(self.course_name))
```

#obj1 is the object of a class Course

obj1 = Course()

#function getinfo() is called using obj1

obj1.getinfo()

Now, these are public variables, therefore, we can change the values.

obj1.fees=4000

obj1.course_name="C++"

obj1.getinfo()

The program window is shown in Figure

```
class Course:
    fees=0
    course_name=""
    def __init__(self):
        self.fees = 3000
        self.course_name="Java"
    def getinfo(self):
        print('course fees is::' + str(self.fees))
        print('course name is::' + str(self.course_name))

#obj1 is the object of a class Course
obj1 = Course()
#function getinfo() is called using obj1
obj1.getinfo()
# Now, these are public variables,therefore, we can change the values.
obj1.fees=4000
obj1.course_name="C++"
obj1.getinfo()
```

Figure 7.13: Modifying the public variable values using object of a class

The output window is shown in Figure

```
course fees is::3000
course name is::Java
course fees is::4000
course name is::C++
>>>
```

Figure 7.14: Output window of Example 7.7 using public access specifier

Conclusion

In this chapter, we discussed OOP fundamentals with the help of illustrative examples. OOP fundamentals discussed with class and objects, the concepts of constructors, inheritance, overriding and encapsulation are explained with simple examples.

In the next chapter, we will cover the concepts of regular expressions, data and time functions in Python.

Points to remember

A class is a template for creating objects.

__init__ is a special method used in constructors to initialize the attributes of an object. It is called automatically, moment an object of a class is created.

In inheritance, child class inherits the properties of the base class.

In method overriding, when a method in a child class shares the same name, parameters, and return type as a method in its parent class, it is referred to as overriding the parent class method.

Encapsulation is the wrapping up variables and methods of a class providing data security. It puts restrictions on the direct access of the data using public, private, and protected access specifiers.

Exercises

What are classes and objects? Discuss with the help of suitable example.

What do you mean by __init__() function? Give example.

Explain the concept of method overriding with the help of suitable example.

What is encapsulation? With the help of example, explain the concept of private and public access specifiers.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 8

Regular Expressions, Date and Time

Introduction

Regular expressions are used to identify patterns in a sequence of strings. To work with the regular expressions in Python, we need to import the re module. The regular expressions help us in searching, matching and manipulating strings based on specific patterns. The datetime module in Python is widely utilized for manipulating dates and times. The current date and time can be obtained using datetime.now() function. You may get the current date with date.today() function.

Structure

The chapter covers the following topics:

Regular expression for pattern matching

Date and time functions

Objectives

By the end of this chapter, you will learn about the regular expression concepts for pattern matching. Apart from this, predefined functions of date time are discussed with the help of examples.

Regular expression for pattern matching

For regular expressions, we use regex or regexp in Python. These are the character combinations that make up a search pattern. They are employed in strings to match patterns. There are certain pre-defined functions associated with the re module as mentioned below:

It finds a first match after looking for a pattern in a string.

It determines whether a pattern at the start of a string matches.

It determines whether a string's starting pattern matches.

In Example re module is used to work with the regular expressions.

Example 8.1: Write a program for substring function using regular expression

Here import re means to import regular expressions. In this example, the substring function is used to replace yes with In this function, y and Y both characters are checked and Yes strings are replaced with a no string.

Code 8.1:

```
import re

str1 = "yes I said yes I will Yes."

res = re.sub("[yY]es","no", str1)

print (res)
```

The program window of Example 8.1 is shown in Figure

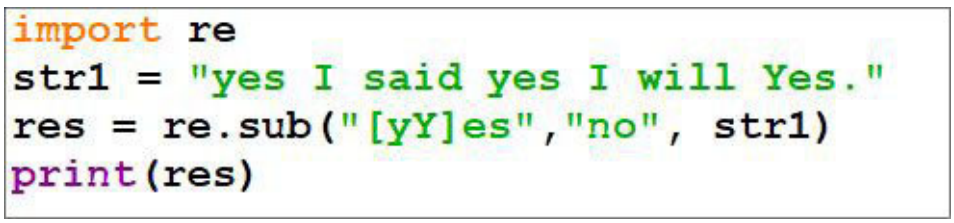


Figure 8.1: Example of regular expression

The output of Example 8.1 is shown in Figure

```
>>>
============ RESTART: E:/python revised/pythonprograms/example8.1.py
no I said no I will no.
>>>
>>>
```

Figure 8.2: Output window of Example 8.1

String searches are frequently conducted using regular expressions. The re module's search() function is used in Example 8.2 to find a pattern's first occurrence in a string.

Example 8.2: Regular expression program to search string

In Example the search() function is used. If the str variable matches with the characters in the string sequence given that is the output will be destination is else, Unable to

Code 8.2:

```
import re

str1 = "The destination is A!"

find = re.search("destination is .*(A|B|C|D)",str1)

if find:

    print(find.group())

else:

    print("Unable to search")
```

The program window of Example 8.2 in IDLE is shown in Figure

```
import re
str1 = "The destination is A!"
find = re.search("destination is .*(A|B|C|D)",str1)
if find:
        print(find.group())
else:
        print("Unable to search")
```

Figure 8.3: Program window of Example 8.2

The output of the Example 8.2 is shown in Figure

```
>>>
============ RESTART: E:/python revised/pythonprograms/example8.2.py
destination is A
>>>
>>>
>>>
```

Figure 8.4: Output window of Example 8.2

The match() function in Python's re module is used to determine whether the regular expression pattern at the start of a string matches or not. In particular, the match function looks for a match at the start of the string. However, we use the search() function if we want to look for a pattern anywhere in the string.

Example 8.3: Write a program to check email ID pattern using the function

In this example, an email is an input from the user in a variable emailid and a function is defined In this function, with the help of the given email ID will be checked with the expression, and the boolean value will be returned from a function after checking the given email ID. If the email provided is valid, the function will return Else, it will return The email entered will be compared to the expression, and if they do not match, an error message stating that the email address is Invalid will appear.

Code 8.3:

```
import re

def checkemail(emailid):

# regular expression pattern for a valid email address

    expression = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'

# Using re.match to check email pattern in the expression

    if re.match(expression,emailid):

        return True

    else:
```

```
    return False


# Input email address

emailid = input("Input an email address:")

# Function checkemail() is called and emailid is passed as parameter

if checkemail(emailid):

    print(emailid, " ","is"," ","Valid")

else:

    print(emailid," ","is"," ","Invalid")
```

The program window is shown in Figure

```
import re
def checkemail(emailid):
# regular expression pattern for a valid email address
    expression = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$'
# Using re.match to check email pattern in the expression
    if re.match(expression,emailid):
        return True
    else:
        return False
# Input email address
emailid = input("Input an email address:")
# Function checkemail() is called and emailid is passed as parameter
if checkemail(emailid):
    print(emailid, " ","is"," ","Valid")
else:
    print(emailid," ","is"," ","Invalid")
```

Figure 8.5: Regular expression to check the email address

The output of Example 8.3 is shown in Figure

Figure 8.6: Output of validating email address using regular expression

The Python re module's findall() function is used in Example 8.4 to find every instance of a pattern in a given text and return them as a list of strings. Every item in the list represents a match.

Example 8.4: Write a program for validating string values only

In this example, fname is a It is checked in a given expression. In this expression, only character value is checked in upper and lower cases, and regular expression will match the pattern using the findall() function, which is a part of the re module. If the input value is not a string, then the message will be displayed to input the correct value.

Code 8.4:

```
import re

# Regular expression for checking only string value

exp= r'\b[A-Za-z]+\b'

fname = input("Enter Your Name")

if(re.findall(exp,fname)):

    print("Welcome:",fname)

else:

    print("Wrong Input,Enter only string value")
```

Note: findall() is a function of re module and returns a list of pattern match in a given input string.

The program window is shown in Figure

```
import re
# Regular expression for checking only string value
exp= r'\b[A-Za-z]+\b'
fname = input("Enter Your Name")
if(re.findall(exp,fname)):
    print("Welcome:",fname)
else:
    print("Wrong Input,Enter only string value")
```

Figure 8.7: Validating a string value using findall() function of re module

The output window is shown in Figure

```
>>>
    = RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python
    Enter Your Name Ramesh
    Welcome:  Ramesh
>>>
    = RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python
    Enter Your Name567
    Wrong Input,Enter only string value
>>>
```

Figure 8.8: Output window of validating only string values using regular expression

Example 8.5: Write a program for validating numerical values only

In Example findall() function is used to check the digit values in a given expression. In this example, rollno is a variable. It is checked in the given expression. In this expression, only numerical value is checked in the given expression. It will match the pattern using findall() function which is a part of re module. If the input value is not integer, then message will be displayed to input correct value.

Code 8.5:

```
import re

# Regular expression for checking only integer value

exp= r'\b[0-9]+\b'

rollno = input("Enter Your Rollno")

if(re.findall(exp,rollno)):

    print("Welcome:",rollno)

else:

    print("Wrong Input,Enter only integer value")
```

The program window is shown in Figure

```
import re
# Regular expression for checking only integer value
exp= r'\b[0-9]+\b'
rollno = input("Enter Your Rollno")
if(re.findall(exp,rollno)):
    print("Welcome:",rollno)
else:
    print("Wrong Input,Enter only integer value")
```

Figure 8.9: Validating only integer values using regular expression

The output of Example 8.5 is shown in Figure

```
Enter Your Rollnoajay
Wrong Input,Enter only integer value
>>>
========== RESTART: E:/python revised/p
Enter Your Rollno1234
Welcome: 1234
```

Figure 8.10: Output window of the integer values using regular expression

Example 8.6: Regular expression to validate the mobile number digits

In this example, a function is defined to check the mobileno of 10 digits using a regular expression. For checking the pattern, re.match() is used. If the mobileno matches within the 10 numbers, then function mobile_validate() will return a True value else it will return a False value.

Code 8.6:

```
import re

# function is defined here

def mobile_validate(mobileno):

# Regular Expression pattern for Mobile Number is defined

    exp = r'^\d{10}$'

# Check the Mobile no.digits using re.match()

    if re.match(exp,mobileno):

return True

    else:
```

```
        return False

mobileno=input("Enter Your Mobile Number")

if mobile_validate(mobileno):

    print(f"{mobileno} is a valid Mobile number.")



else:

    print(f"{mobileno} is not a valid Mobile number.")
```

The program window is shown in Figure

```
import re
# function is defined here
def mobile_validate(mobileno):
# Regular Expression pattern for Mobile Number is defined
    exp = r'^\d{10}$'
# Check the Mobile no.digits using re.match()
    if re.match(exp,mobileno):
        return True
    else:
        return False

mobileno=input("Enter Your Mobile Number")
if mobile_validate(mobileno):
    print(f"{mobileno} is a valid Mobile number.")
else:
    print(f"{mobileno} is not a valid Mobile number.")
```

Figure 8.11: Program to validate the mobile number digits

The output window is shown in Figure

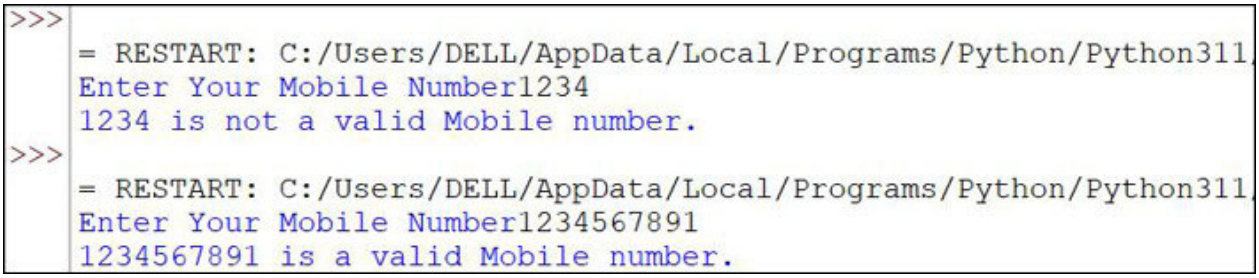


Figure 8.12: Output of validating a regular expression for mobile number

Date and time functions

Python offers numerous modules and functions for handling time and dates. Datetime, time, and calendar are the modules that are most frequently used when working with dates and times. Here are a few Python datetime function examples.

Example 8.7: Write a program to display time

In this example, an import statement is used to import time package and pre-defined functions of time are called.

Code block 8.7:

```
import time;

localtime = time.asctime( time.localtime(time.time()) )

print("Local current time :", localtime)
```

The program window of Example 8.7 in IDLE is shown in Figure

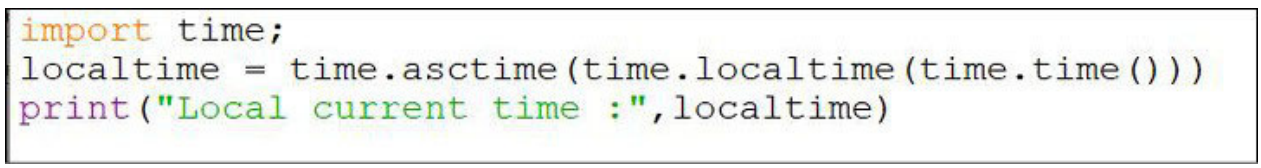


Figure 8.13: Program window of Example 8.7

The output of Example 8.7 is shown in Figure

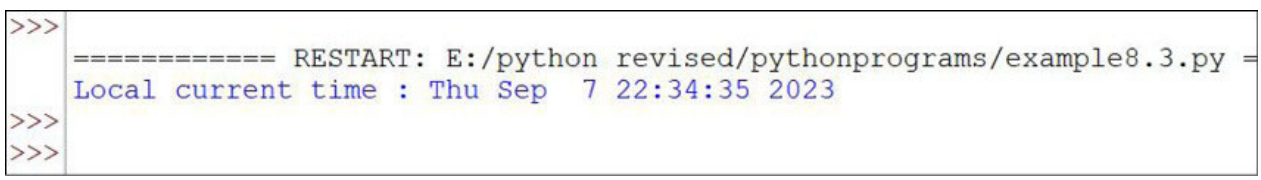


Figure 8.14: Output window of Example 8.7

Example 8.8: Write a program to display datetime using now function

In this example, the datetime package is imported using an import statement. A predefined function now() for displaying date and time is called.

Code 8.8:

```
import datetime

print(datetime.datetime.now())
```

The program window of Example 8.8 in IDLE is shown in Figure

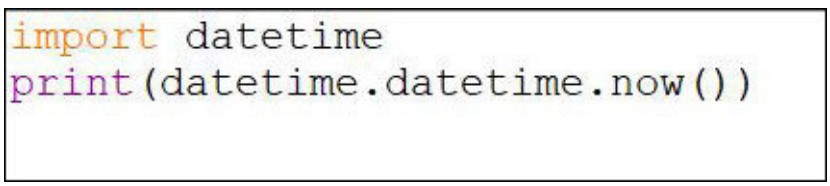


Figure 8.15: Program window of Example 8.8

The output of Example 8.8 is shown in Figure

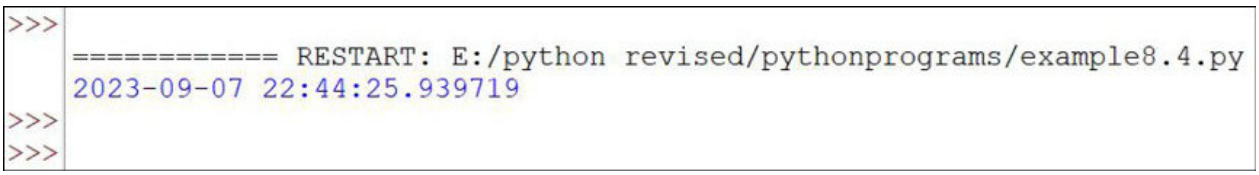


Figure 8.16: Display date and time using now () function

Example 8.9: Write a program to obtain the calendar month using calendar module

In this example, a calendar module is used, and using the month() function, the specific month is obtained.

Code 8.9:

```
import calendar

# calender for a specific month is obtained

cal_month = calendar.month(2022,9)

print(cal_month)
```

The program window is shown in Figure

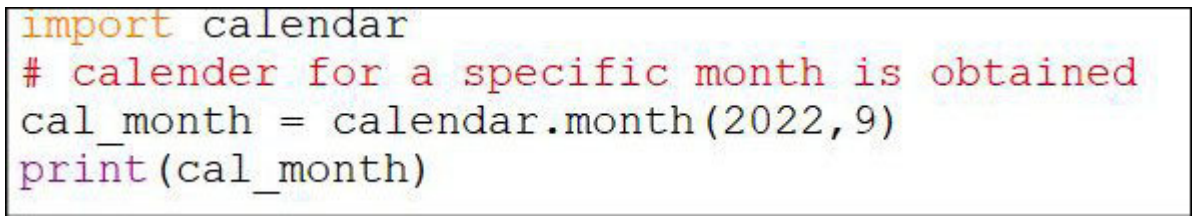


Figure 8.17: Program window using calendar module

The output window is shown in Figure

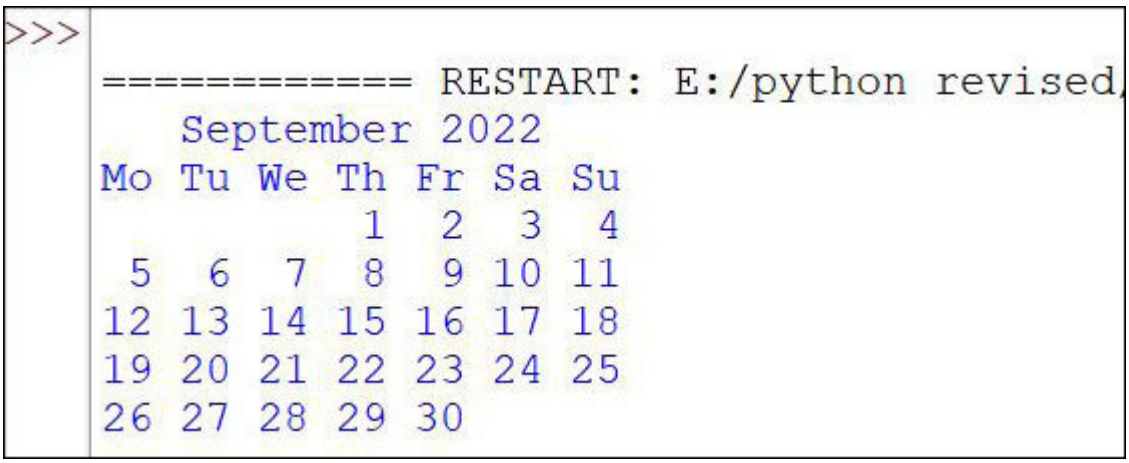


Figure 8.18: Output window of calendar module

Conclusion

In this chapter, regular expressions are discussed for pattern matching and text manipulation. Here, the re module is imported to work with regular expressions. Regular expression operations are implemented using pre-built functions for various text processing tasks like searching, matching, replacing, and more. In addition, datetime preset functions are shown using simple illustrations.

In the next chapter, the fundamentals of exception handling are discussed. In Python, addressing exceptions and errors that may arise while a program is running is known as exception To manage exceptions, Python has the try, except block to monitor the statements that may raise exceptions.

Points to remember

Python’s regular expressions, also known as regex, are an effective tool for text manipulation and pattern matching.

The re module can be used to manipulate regular expressions. Regular expressions can be used for a number of text processing operations, including searching, matching, replacing, and more.

For various regex operations, the re module offers functions like and

You can use re.search() to locate the pattern’s first match within a string and re.match() to determine whether the pattern matches at the string’s beginning.

When a pattern appears in a string, re.sub() can be used to replace each instance with a particular value.

To work with date and time functions, Python offers the datetime module.

The time module provides time-related functions like measuring intervals of time.

The datetime module provides a more comprehensive way to work with date and time than the time module does.

Calendar-related classes and operations, including printing and formatting calendars, are provided by the calendar module.

Exercises

What do you mean by regular expression?

Write the pre-defined functions associated with regular expressions.

Write a program for regular expression to search a string.

What do you mean by validation? Give example of validating string values.

What are date and time functions? Give suitable example.

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 9

Exception Handling

Introduction

Exception is an error that occurs during the execution of the program. Exception handling provides a user-friendly interface to manage these errors and presents a clear message to the user explaining the reason for such unexpected occurrences. For example, attempting to divide a number by zero would prompt an exception.

Structure

In this chapter, the following topics are covered:

Exception error occurred due to inappropriate values

Exception error raised due to different data types

IOError exception

ModuleNotFound exception

ZeroDivisionError exception

Objectives

By the end of this chapter, learners will be able to apply the concept of exception handling in programs. The reasons for exception errors occurring in programs are discussed with the help of suitable examples.

Exception

An exception is an event that happens while a program is being executed, and a program's execution encounters an undesirable or unexpected event known as an exception. Exception handling is a vital part of programming that improves programs' durability, dependability, and user experience by effectively dealing with unforeseen circumstances and mistakes.

Errors like file not found or corrupted files could occur when reading or writing to files. Errors may arise from various sources, such as network problems, failed database connections, incorrect user input, hardware failures, mathematical exceptions, or problems with third-party libraries.

In this chapter, exceptions that occurred due to different errors are discussed with the help of examples.

An exception error arises in Example 9.1 as a result of improper input, raising an exception. Common exceptions for incorrect values are errors such as and AssertionError among others. When a function receives an argument with the correct datatype but an invalid value, ValueError is raised. Usually, it signifies that the input value is inappropriate for the task at hand.

When a function is applied to an object that is not of the appropriate type, a TypeError exception is raised. When an object's data type is incompatible with the operation or function being utilized, an error occurs.

When an assert statement fails, an AssertionError takes place. To verify whether a condition is true for debugging reasons, use the assert statement. An AssertionError is raised if the condition is false.

Example 9.1: Program of exception error occurred due to inappropriate values

In this example, a variable no will store the value entered, and if the input value is not a number, then an exception will be raised, and a message will be printed that says Enter only

Code 9.1:

```
import sys

try:

no = int(input("Enter a number"))

except ValueError:

  print("Enter only numbers")

  sys.exit()

print ("you entered number",no)
```

The program window in IDLE is shown in Figure

```
import sys
try:
    no = int(input("Enter a number"))
except ValueError:
    print("Enter only numbers")
    sys.exit()
print("you entered number",no)
```

Figure 9.1: Exception error occurred due to inappropriate values

Note: Try and except are Python keywords to monitor exceptions that occur in the programs.

ValueError is raised when an inappropriate value is passed; like in Example the string is entered in place of an integer as a value.

The output of Example 9.1 is shown in Figure

```
>>>
    ========== RESTART: E:/python revised/pythonprograms/example9.1.py
    Enter a numberABC
    Enter only numbers
>>>
    ========== RESTART: E:/python revised/pythonprograms/example9.1.py
    Enter a number 45
    you entered number 45
>>>
>>>
```

Figure 9.2: ValueErrorexception

Example 9.2: Program of exception error raised due to different data types

This example will raise an exception for the wrong data type. In variable integer and string values are added, which results in exception.

Code 9.2:

```
import sys

try:

    a =5

    b = 'ABC'

c = a + b

except TypeError:

    print('Caught Exception')
```

The program window of Example 9.2 in IDLE is shown in Figure

```
import sys
try:
    a =5
    b = 'ABC'
    c = a + b
except TypeError:
    print ('Caught Exception')
```

Figure 9.3: Example of DataTypeerror

Note: TypeError is raised when we try to add integer and string values, which are of different data types.

The output of Example 9.2 is shown in Figure

```
>>>
>>>
>>>
========== RESTART: E:/python revised/pythonprograms/example9.2.py
Caught Exception
>>>
```

Figure 9.4: Output of TypeErrorexception

Example 9.3: Program of IOError exception

In this program, the IOError exception comes when the mode is changed from that is, write mode, to that is, read mode of the file. If file modes are proper, the program runs successfully and writes the data into the file In order to run this example, we must create a text file called test2.txt on any drive. This is because a file must exist in order to write data into it; otherwise, an error will occur.

Code 9.3:

```
try:

  fo = open("C:\\test2.txt", "w")

  fo.write("This is test file")

except IOError:

  print ("Error: enable to find file or read data")

else:

print ("data written in the file successfully")

fo.close()
```

The program window of Example 9.3 in IDLE is shown in Figure

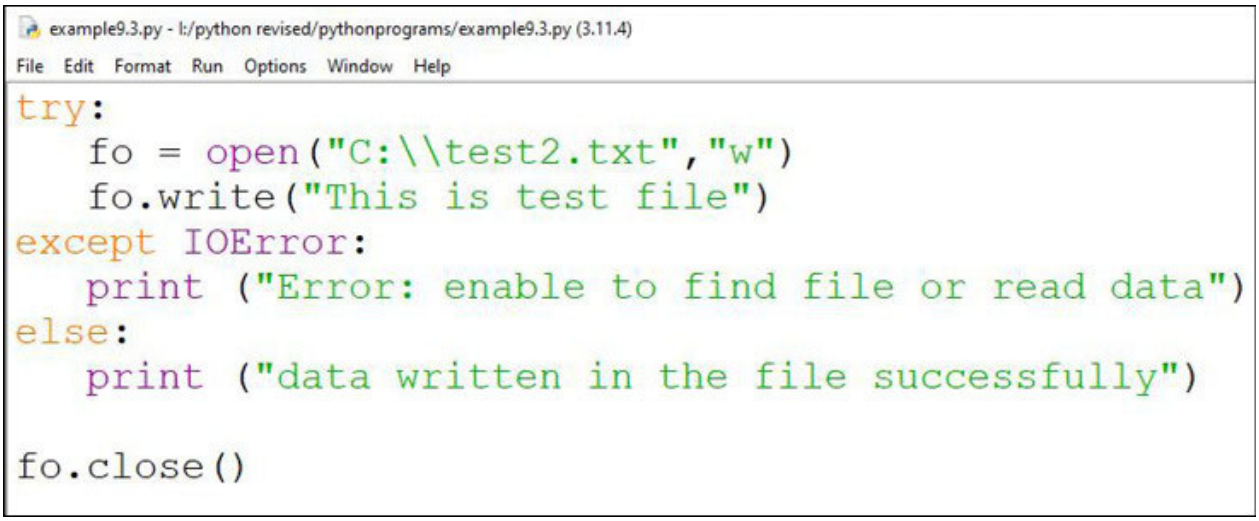

example9.3.py - I:/python revised/pythonprograms/example9.3.py (3.11.4)
File Edit Format Run Options Window Help
try:
fo = open("C:\\test2.txt","w")
fo.write("This is test file")
except IOError:
print ("Error: enable to find file or read data")
else:
print ("data written in the file successfully")
fo.close()

Figure 9.5: Program window of Example 9.3

The output of Example 9.3 is shown in Figure

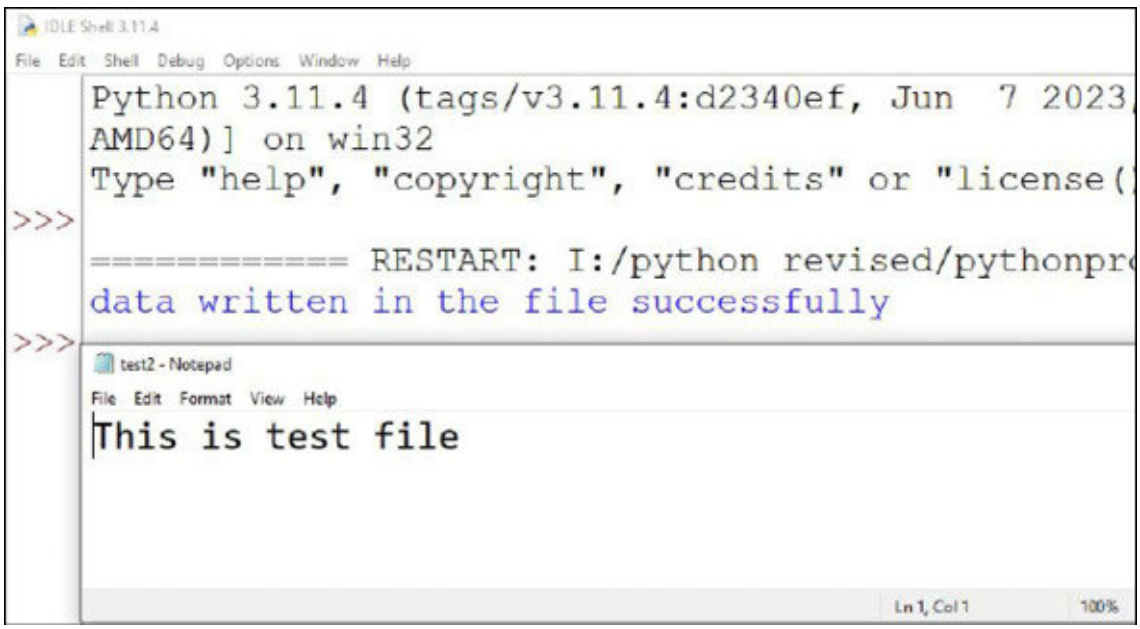


Figure 9.6: Data is written successfully into a text file named test2.txt

Note: On changing the “mode” from “w” to “r”, the Exception is raised, that is, “enable to find file or read data” as shown in Figure

Example 9.4: Program of ModuleNotFound exception

In this example, an exception will be raised when a module is not found. Therefore, first of all, create a module and save it as test.py as shown in Figure and import it in as shown in Figure

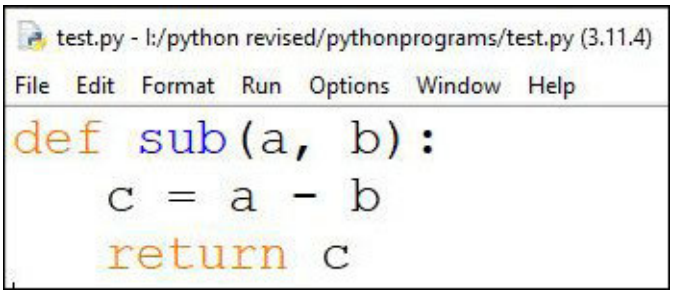


Figure 9.7: Create a module named test

The module name is saved as

Code 9.4:

```
def sub(a, b):

  c = a - b

  return c
```

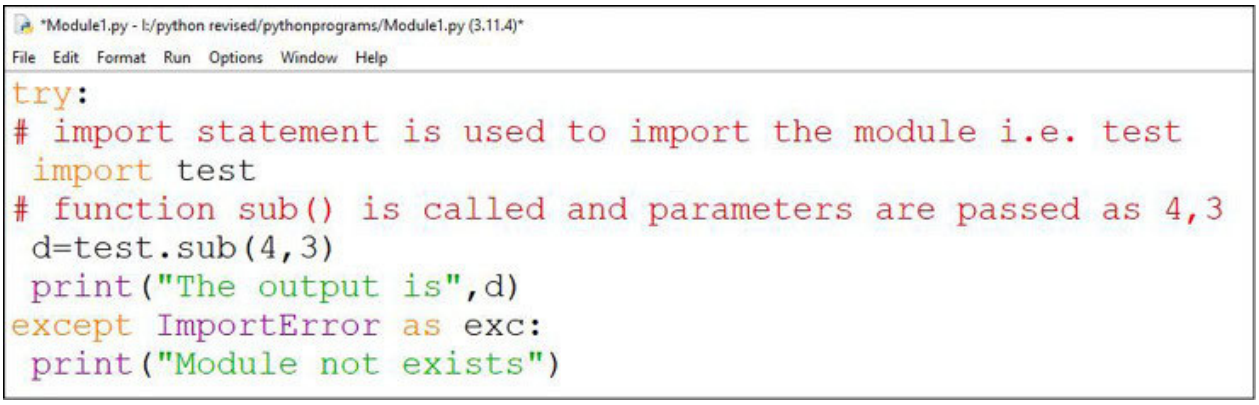

Module1.py - I:/python revised/pythonprograms/Module1.py (3.11.4)
File Edit Format Run Options Window Help
try:
# import statement is used to import the module i.e. test
import test
# function sub() is called and parameters are passed as 4,3
d=test.sub(4,3)
print("The output is",d)
except ImportError as exc:
print("Module not exists")

Figure 9.8: Module test is imported in Module1.py

Now the above module test is imported in Module1.py mentioned in Code

try:

import statement is used to import the module i.e. test

import test

function sub() is called and parameters are passed as 4,3

d=test.sub(4,3)

print ("The output is",d)

except ImportError as exc:

print ("Module not exists")

In the module test is imported in which the sub() function is defined, and the values 4 and 3 are passed as parameters in sub(a, The values are passed as parameters, and the result will be stored and returned in a variable which shows the output as 1, but on changing the name of the module, an exception occurs, and the output is shown as Module not The output of Example 9.4 is shown in Figure

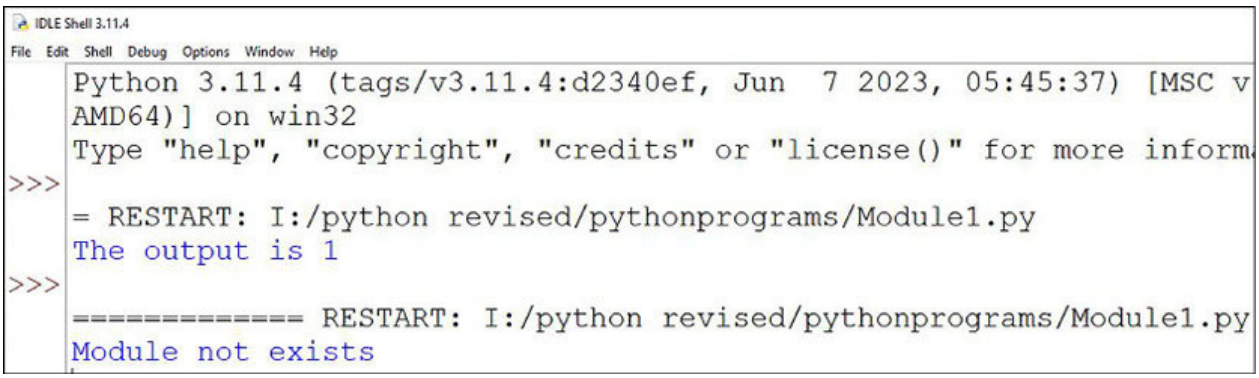

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more inform
>>>
= RESTART: I:/python revised/pythonprograms/Module1.py
The output is 1
>>>
========== RESTART: I:/python revised/pythonprograms/Module1.py
Module not exists

Figure 9.9: Output of Example 9.4

In Figure test module is imported correctly and in another case, an exception is raised when test module is not found, that is, Import Error

Example 9.5: Write a program for division by zero exception

In this example, a number is divided by zero, which results in an exception of The program window of Example 9.5 is shown in Figure

Code 9.5:

try:

    output = int(input("Enter a number"))

    output1 = output/0

ZeroDivisionError Exception

except ZeroDivisionError:

    print("This is an exception of dividing a value by zero")

Refer to the following figure:

```
try:
    output = int(input("Enter a number"))
    output1 = output/0
# ZeroDivisionError Exception
except ZeroDivisionError:
    print("This is an exception of dividing a value by zero")
```

Figure 9.10: Program window of division by zero exception

The output window is shown in Figure

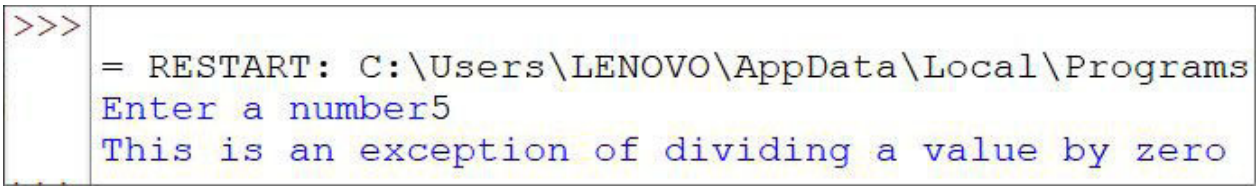


Figure 9.11: Output window of division by zero exception

Example 9.6: Example of assert statement for exception handling

In Python, the assert statement is used for debugging and works like the if-else condition. In this example, a function func_divide() is defined, and the value is passed as an input in variables var1 and The function func_divide() will not raise an exception if the values passed are correct; otherwise, if the number passed as a parameter is zero, the number will be divided by zero, and an AssertionError will appear with a message indicating that Exception of Division by Zero has occurred.

Code 9.6:

```
def func_divide(var3,var4):

assert var4!=0,"Exception of Division by zero"

    return var3/var4

var1 = int(input("Enter First Value"))

var2 = int(input("Enter Second Value"))

output = func_divide(var1,var2)

print(output)
```

The program window is shown in Figure

```
def func_divide(var3,var4):
    assert var4!=0,"Exception of Division by zero"
    return var3/var4

var1 = int(input("Enter First Value"))
var2 = int(input("Enter Second Value"))
output = func_divide(var1,var2)
print(output)
```

Figure 9.12: Program window of Assertionerror of division by zero

The function func_divide() returns a valid value in the output window depicted in Figure 9.13 when the numbers supplied are integer values. However, if the values passed contain zero values, the assert statement will produce an error pertaining to the Divide by Zero exception:

```
>>>
    = RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/example9.6.py
    Enter First Value25
    Enter Second Value2
    12.5
>>>
    = RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/example9.6.py
    Enter First Value25
    Enter Second Value0
    Traceback (most recent call last):
      File "C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/example9.6.py",
    line 7, in <module>
        output = func_divide(var1,var2)
      File "C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/example9.6.py",
    line 2, in func_divide
        assert var4!=0,"Exception of Division by zero"
    AssertionError: Exception of Division by zero
>>>
```

Figure 9.13: Output window of Assert error

Note: ‘assert’ is a keyword in Python used for debugging, testing, and working like an ‘if-else’ condition for raising an ‘assertionexception’ error if the condition doesnot fulfill the criteria.

Conclusion

In Python, addressing exceptions is a crucial aspect of developing reliable programming. It enables you to respond to exceptional errors and recover from them with responsiveness. In this chapter, different exceptions, like exceptions due to inappropriate values, data type errors, ModuleNotFound exceptions, and are discussed with the help of suitable examples.

In the next chapter, we will discuss the NumPy and Tkinter fundamentals.

Points to remember

Try and except are Python keywords to monitor exceptions that occur in the programs.

The ValueError exception error is raised when an inappropriate value is passed.

TypeError exception is raised when we try to add integer and string values of different types.

IOError is an input and output error related to

ImportError exception occurs when the modulename used is incorrect, or the module is not found in a program.

ZeroDivisionError occurs when we try to divide a number by zero.

assert is a keyword in Python used for debugging testing and works like an if-else condition for raising an assertionexception error if the condition does not fulfill the criteria. It includes exceptions like the DivisionByZero exception.

Exercises

What do you mean by exception in Python, and how do we handle it?

How does an exception occur due to inappropriate values?

What is an IOError exception?

Explain the ModuleNotFound exception and ZeroDivisionError exception with the help of a suitable example.

Explain the purpose of using the try, except, and finally blocks in exception handling.

What is the difference between a runtime error and a syntax error in Python?

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

CHAPTER 10

NumPy and Tkinter

Introduction

Numerical Python a pivotal open-source Python library, revolutionizes mathematical and numerical computations with support for multidimensional arrays and elementary operations. Widely utilized across domains like data analysis and scientific research, NumPy is indispensable for scientific computing. Tkinter, another featured topic, serves as a premier Python framework for crafting intuitive GUI-based desktop applications. With Tkinter's user-friendly toolkit, developers effortlessly design windows, buttons, and other GUI elements, enhancing user experience. Together, NumPy and Tkinter epitomize Python's versatility in facilitating both robust numerical computations and seamless GUI development.

Structure

The topics covered in this chapter are:

1-D array operations using NumPy

Matrix addition and multiplication using NumPy

Fundamentals of Tkinter main application window

Label and button widget in Tkinter

Button click event and input from user in Tkinter

Objectives

By going through this chapter, learners will acquire the knowledge of NumPy and perform operations on arrays using NumPy. Apart from that, you will be able to work with a GUI-based environment of Python known as Tkinter.

Note: In order to install NumPy, first of all, open the Windows command prompt, navigate to the Python installation directory, and open the script folder. Now, using cd, that is, change directory command, type 'pip install numpy' as shown in Figure

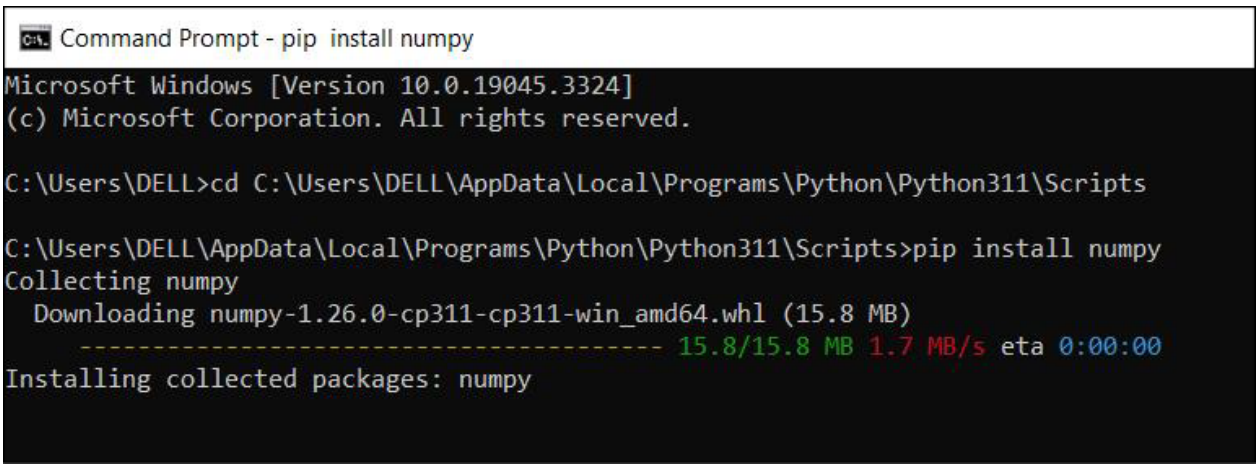

Command Prompt - pip install numpy
Microsoft Windows [Version 10.0.19045.3324]
(c) Microsoft Corporation. All rights reserved.
C:\Users\DELL>cd C:\Users\DELL\AppData\Local\Programs\Python\Python311\Scripts
C:\Users\DELL\AppData\Local\Programs\Python\Python311\Scripts>pip install numpy
Collecting numpy
Downloading numpy-1.26.0-cp311-cp311-win_amd64.whl (15.8 MB)
15.8/15.8 MB 1.7 MB/s eta 0:00:00
Installing collected packages: numpy

Figure 10.1: Installing the NumPy library using the pip command

1-D array operations using NumPy

For numerical and mathematical operations in Python, NumPy is an invaluable tool, especially when working with arrays. Here, we use an array object to execute various actions on arrays, including generating arrays, accessing array items, element-wise functions, arithmetic operations, sorting, filtering, and so on. A 1D array is built in Example

Example 10.1: Write a program to create a simple 1D array from a Python list

In this example, a NumPy package is imported, and an array a1 is created. The value at the index is accessed. Therefore, the output is 2.

Code 10.1:

#In this example, a numpy package is imported and a 1-Dimensional array

#is created from a Python List

import numpy as np

Creating a 1-Dimentional array from a Python list

a1 = np.array([1, 2, 3, 4, 5])

print(a1[1])

The program window of Example 10.1 is shown in Figure

```
#In this example, a numpy package is imported and a 1-Dimensional array
#is created from a Python List

import numpy as np
# Creating a 1-Dimentional array from a Python list
a1 = np.array([1, 2, 3, 4, 5])
print(a1[1])
```

Figure 10.2: Example of 1-D array using NumPy package

The output window is shown in Figure

Figure 10.3: Output of 1-D array using NumPy

Example 10.2 shows how to construct built-in functions or libraries to operate on lists or arrays using the sum, mean, max, and min functions. Since NumPy is specialized for numerical operations and uses vectorization, it is frequently more efficient when working with huge datasets or arrays. NumPy also offers various statistical operations and functions that are helpful in a variety of situations.

Example 10.2: Finding sum, mean, max, and min functions using NumPy

In this example, the basic operations like sum, mean, maximum, and minimum are implemented using pre-defined functions in NumPy.

Code 10.2:

```
import numpy as np

# Creating a NumPy array

a1 = np.array([3,4,5,6,7])

print("NumPy array operations using pre-defined functions:")

print(a1)

# Finding sum,mean,max and min in Numpy

print("\nBasic Operations:")

print("Sum of array:", np.sum(a1))

print("Mean of array:", np.mean(a1))

print("Max of array:", np.max(a1))
```

print("Min of array:", np.min(a1))

The program window is shown in Figure

```
import numpy as np
# Creating a NumPy array
a1 = np.array([3,4,5,6,7])
print("NumPy array operations using pre-defined functions:")
print(a1)
# Finding sum,mean,max and min in Numpy
print("\nBasic Operations:")
print("Finding Sum of array:", np.sum(a1))
print("Finding Mean of array:", np.mean(a1))
print("Finding Max of array:", np.max(a1))
print("Finding Min of array:", np.min(a1))
```

Figure 10.4: NumPy basic operations on arrays

The output of Example 10.2 is shown in Figure

```
NumPy array operations using pre-defined functions:
[3 4 5 6 7]

Basic Operations:
Sum of array: 25
Mean of array: 5.0
Max of array: 7
Min of array: 3
```

Figure 10.5: Output of pre-defined functions using arrays in NumPy

Matrix addition and multiplication using NumPy

Matrix addition and multiplication are done in Example Its crucial to remember that matrices can also be added to and multiplied using the NumPy np.add() and np.dot() operations.

Example 10.3: Write a program for matrix addition and multiplication using NumPy

In this example, matrix addition and multiplication operations are performed on two arrays using NumPy. The program window in IDLE is shown in Figure

```
import numpy as np
print("\nMatrix Addition and Multiplication operations :")
# array a1 is created
a1 = np.array([5, 6, 7])
# array a2 is created
a2 = np.array([8, 9, 10])
# addition of matrix
output_add = a1+a2
print("Addition of Matrix:", output_add)
# multiplication of matrix
output_mult = a1 * a2
print("Multiplication of Matrix:", output_mult)
```

Figure 10.6: Matrix addition and multiplication using NumPy

Code 10.3:

import numpy as np

print("\nMatrix Addition and Multiplication operations :")

array a1 is created

a1 = np.array([5, 6, 7])

```
# array a2 is created

a2 = np.array([8, 9, 10])

# addition of matrix

output_add = a1+a2



print("Addition of Matrix:", output_add)

# multiplication of matrix

output_mult = a1 * a2

print("Multiplication of Matrix:", output_mult)
```

The output window of Example 10.3 is shown in Figure

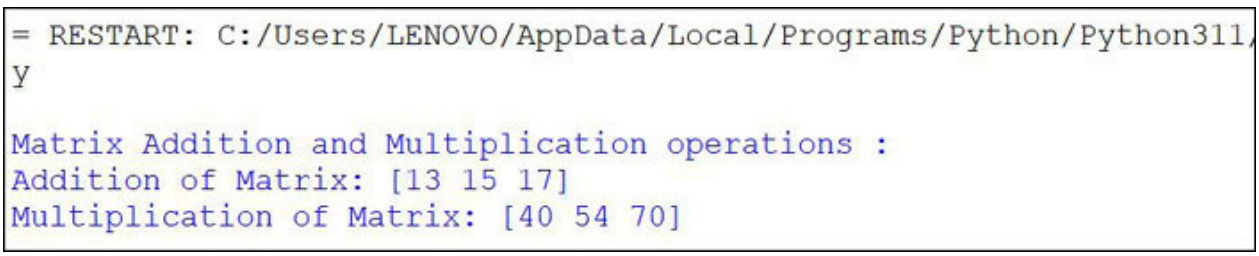

= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python311/
y
Matrix Addition and Multiplication operations :
Addition of Matrix: [13 15 17]
Multiplication of Matrix: [40 54 70]

Figure 10.7: Output window of matrix addition and multiplication using NumPy

Fundamentals of Tkinter main application window

The standard Python Graphical User Interface toolkit is called Tkinter. It is based on the Tk GUI toolkit and offers a set of tools for designing graphical user interfaces. The Tk class is used to construct the main application window. To utilize the Tkinter module's classes and functions, import the module

Example 10.4: Example of Tkinter application window

In this example, Tkinter is to be imported in a Python script. Tk() is a class and an instance of this class is created to show the main application window. Root is a variable for the main application window, that is, the root window.

Code 10.4:

```
#We need to import tkinter in a python script

import tkinter as tk

#Here Tk() is the class and we need to create the

#instance of this class for creating a main application window

root = tk.Tk()

#To create a main application window, you need to create an instance of the Tk() class:

root.title("First Example")
```

The program window is shown in Figure

```
#We need to import tkinter in a python script
import tkinter as tk
#Here Tk() is the class and we need to create the
#instance of this class for creating a main application window
root = tk.Tk()
#To create a main application window, you need to create an instance of the Tk()class:
root.title("First Example")
```

Figure 10.8: Tkinter main application window

The output is shown in Figure

Figure 10.9: Output window of Tkinter main application window

Label and button widget in Tkinter

A GUI's widgets are its fundamental components. To the main window, we can add different widgets, such as buttons, labels, entry fields, and so on. Text labels and buttons are created in Tkinter using the label and button widgets, respectively. Using a label widget is created, and the text to be shown is specified. Similarly, tk.Button() is used to create a button widget. When the button is clicked, a function is supposed to be called.

Example 10.5: Create a label and button widget in Tkinter

In this example, label and button widgets are created. These widgets are Python objects. The main window is created using

Code 10.5:

```
import tkinter as tk

#Main window is created using tk.Tk()

root = tk.Tk()

root.title("Example of button click")

# A Label widget is created

# Label widget is a Python object

label = tk.Label(root, text="Label Widget")

# pack() method is used to arrange the widgets.

label.pack(pady=10)

# A button widget is created
```

Button widget is a Python Object

button = tk.Button(root, text="Click Here")

button.pack()

root.mainloop()

The program window is shown in Figure

```
import tkinter as tk

#Main window is created using tk.Tk()
root = tk.Tk()
root.title("Example of button click")
# A Label widget is created
# Label widget is a Python object
label = tk.Label(root, text="Label Widget")
# pack() method is used to arrange the widgets.
label.pack(pady=10)
# A button widget is created
# Button widget is a Python Object
button = tk.Button(root, text="Click Here")
button.pack()
root.mainloop()
```

Figure 10.10: A label and button widgets

The output window is shown in Figure

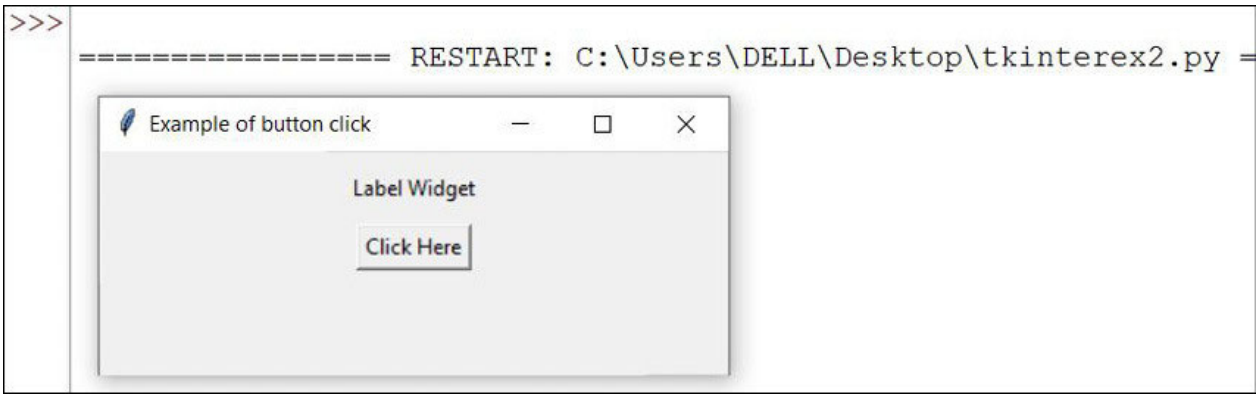


Figure 10.11: Output of label and button widget

Button click event and input from user in Tkinter

With Tkinter, we can utilize a variety of widgets and event handlers to manage button click events and user input. The button click event is linked to a button widget. Example 10.6 defines a function that is called upon button click.

Example 10.6: Example of button click event

In this example, a function btnclick() is defined and is called using the command. This creates a label and button widget and uses the variable root in Tkinter for the main application window. Here, pack() is used to place a widget in a and mainloop() is a method that runs an infinite loop and keeps the GUI running until the user closes the main window. It keeps the application responsive to the user actions.

Code 10.6:

```
#Here tkinter is imported

import tkinter as tk

#btnclick() function is defined

def btnclick():

    label.config(text="You have clicked a button")

#root is a variable used in tkinter for main application window i.e. root window

root = tk.Tk()

# Main window is created
```

```
root.title("Example of a button click event")

# Label widget is created

label = tk.Label(root, text="Example of Button Click Event")



label.pack(pady=10)

# Button widget is created and a function btnclick is called

#command is used with widgets & for interacting when button is clicked and so on.

button = tk.Button(root, text="Click Me", command=btnclick)

#pack() is used to place a widget in a block

button.pack()

#mainloop() is a method that runs an infinite loop and keeps the GUI

# runninguntil the user closes the main window.

#it keeps the application responsive to the user actions.

root.mainloop()
```

The program window is used in Figure

```
# Example of a Button click event

#Here tkinter is imported
import tkinter as tk
#btnclick() function is defined
def btnclick():
    label.config(text="You have clicked a button")
#root is a variable used in tkinter for main application window i.e.root window
root = tk.Tk()
# Main window is created
root.title("Example of a button click event")
# Label widget is created
label = tk.Label(root, text="Example of Button Click Event")
label.pack(pady=10)
# Button widget is created and a function btnclick is called
#command is used with widgets and for interacting when button is clicked etc.
button = tk.Button(root, text="Click Me", command=btnclick)
#pack() is used to place a widget in a block
button.pack()
#mainloop() is a method that runs an infinite loop and keeps the GUI running
#until the user closes the main window.
#it keeps the application responsive to the user actions.
root.mainloop()
```

Figure 10.12: Example of button click event

The output window of the above example is shown in Figure

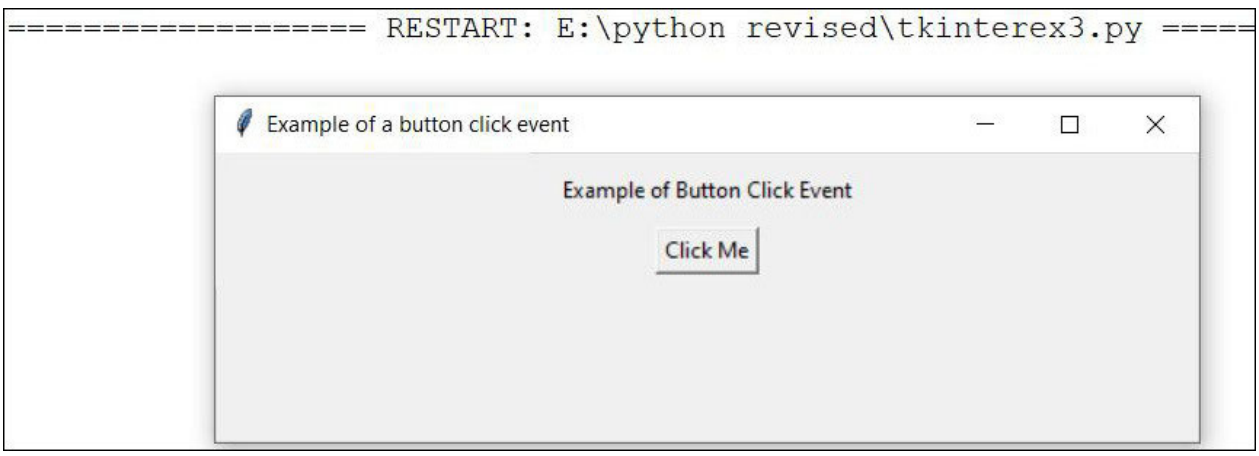


Figure 10.13: Example of button click event

Example 10.7: Example of taking input from a user using the text field

In this example, the root window is created using the root variable. The var1 variable is created for the label widget, the var2 variable is used for the entry widget, that is, used for taking input, and var3 is again used for the label widget, and var4 is for the button. On the click event of the button using the command option, a function get_input() is called, and the input value is taken from var2 on the button, the Welcome message will be displayed using

Code 10.7:

```
import tkinter as tk

# function is defined here

def get_input():

    user = var2.get()

    var4.config(text=f"Welcome: {user}")

#root is a variable used in tkinter for main application window

#i.e. root window

#Tk() is a class and is a container for all the widgets
```

```
root = tk.Tk()

# Main window is created



root.title("Example of taking input from a user")

var1 = tk.Label(root, text="Enter Your Name")

var1.pack(pady=5)

# Entry is also one of the widget and is used for input fields

var2 = tk.Entry(root, width=40)

# Pack() is used for arranging widgets within the window and places widgets
in a block

var2.pack(pady=5)

#Label is a widget and is used for displaying text or images

var3 = tk.Label(root, text="")

#pack() is used to place widgets in a block

var3.pack()

# Here function get_input() is called on button click event using
```

command option and no parentheses are required

#Button is the widget and used for firing event

var4 = tk.Button(root, text="Click Button", command=get_input)

var4.pack()

#mainloop() is a method and runs infinite loop.

#mainloop() keeps the application running until the user closes the

main window.

root.mainloop()

The program window of example is shown in Figure

```
import tkinter as tk
# function is defined here
def get_input():
    user = var2.get()
    var4.config(text=f"Welcome: {user}")

#root is a variable used in tkinter for main application window i.e. root window
#Tk() is a class and is a container for all the widgets
root = tk.Tk()
# Main window is created
root.title("Example of taking input from a user")
var1 = tk.Label(root, text="Enter Your Name")
var1.pack(pady=5)
# Entry is also one of the widget and is used for input fields
var2 = tk.Entry(root, width=40)
# Pack() is used for arranging widgets within the window and places widgets in a block
var2.pack(pady=5)
#Label is a widget and is used for displaying text or images
var3 = tk.Label(root, text="")
#pack() is used to place widgets in a block
var3.pack()
# Here function get_input() is called on button click event using command option and no parentheses are required
#Button is the widget and used for firing event
var4 = tk.Button(root, text="Click Button", command=get_input)
var4.pack()
#mainloop() is a method and runs infinite loop.
#mainloop() keeps the application running until the user closes the main window.
root.mainloop()
```

Figure 10.14: Program window for taking input from user using Tkinter

The output window is shown in Figure

Figure 10.15: Output window of getting input from user in Tkinter

Example 10.8: Write a program for performing arithmetic operations

In this example, arithmetic operations, that is, addition, subtraction, multiplication, and division, are performed. A function arithmetic_operations() is defined for taking input from the user. Here, Tkinter library is imported and a main window is created and its title is set. In this example, widgets like label, entry field, command buttons are created. Pack() method is used for arranging widgets within the window and places widgets in a block. Four command buttons are created for addition, subtraction, division, and multiplication. A function arithmetic_operations() is called inside the command button for passing the parameter into a choice variable. mainloop() is a method and it runs an infinite loop. The mainloop() method keeps the application running until the user closes the main window.

Code 10.8:

```
tkinter as tk #Import a library

def arithmetic_operations(choice): #function is defined here

    try:    # Exception handling is used here

        number_1 = int(num_1.get())#user input is taken here

        number_2 = int(num_2.get())#user input is taken here

        if choice == "Addition":
```

```
            output.set(f"Output is: {number_1+number_2}")


        elif choice == "Subtraction":

            output.set(f"Output is: {number_1-number_2}")

        elif choice == "Multiplication":

            output.set(f"Output is: {number_1*number_2}")

        elif choice == "Division":

            output.set(f"Output is: {number_1/number_2}")

    except ValueError:

            output.set("Entered value is not valid")

#main window is created here and its title is set

root = tk.Tk()

root.geometry("400x400")

root.title("********Arithmetic Operations*********")

#Label1 is set here
```

```
num_1=tk.Label(root,text="Enter First Number",bg="pink",fg="blue")

# Pack() is used for arranging widgets within the window and places

# widgets in a block

num_1.pack()



# entry field1 for user input is created here

num_1 = tk.Entry(root,width=20)

num_1.pack()

#Label2 is set here

num_2=tk.Label(root,text="Enter Second Number",bg="pink",fg="blue")

num_2.pack()

# entry field2 for user input is created here

num_2 = tk.Entry(root,width=20)

num_2.pack()

# 4 buttons are created for addition,subtraction,division and multiplic
```

```
# ation.A function arithmetic_operations() is called here

addition = tk.Button(root, text="Addition", command=lambda:
arithmetic_operations("Addition"),width =20)

subtraction = tk.Button(root, text="Subtraction", command=lambda:
arithmetic_operations("Subtraction"),width =20)

multiplication = tk.Button(root, text="Multiplication", command=lambda:
arithmetic_operations("Multiplication"),width =20)



division = tk.Button(root, text="Division", command=lambda:
arithmetic_operations("Division"),width =20)

addition.pack()

subtraction.pack()

multiplication.pack()

division.pack()

# Label created to display the output

output = tk.StringVar()

output_label = tk.Label(root,textvariable=output)
```

output_label.pack()

#mainloop() is a method and runs infinite loop.#mainloop() keeps the

application running until the user closes the main window.

root.mainloop()

A program window is shown in Figure

```
import tkinter as tk #Import a library
def arithmetic_operations(choice): #function is defined here
    try:    # Exception handling is used here
        number_1 = int(num_1.get())#user input is taken here
        number_2 = int(num_2.get())#user input is taken here
        if choice == "Addition":
            output.set(f"Output is: {number_1+number_2}")
        elif choice == "Subtraction":
            output.set(f"Output is: {number_1-number_2}")
        elif choice == "Multiplication":
            output.set(f"Output is: {number_1*number_2}")
        elif choice == "Division":
            output.set(f"Output is: {number_1/number_2}")
    except ValueError:
            output.set("Entered value is not valid")
#main window is created here and its title is set
root = tk.Tk()
root.geometry("400x400")
root.title("********Arithmetic Operations*********")
#Label1 is set here
num_1=tk.Label(root,text="Enter First Number",bg="pink",fg="blue")
# Pack() is used for arranging widgets within the window and places widgets in a block
num_1.pack()
# entry field1 for user input is created here
num_1 = tk.Entry(root,width=20)
num_1.pack()
#Label2 is set here
num_2=tk.Label(root,text="Enter Second Number",bg="pink",fg="blue")
num_2.pack()
# entry field2 for user input is created here
num_2 = tk.Entry(root,width=20)
num_2.pack()
# 4 buttons are created for addition,subtraction,division and multiplication.A function arithmetic_operations() is called here
addition = tk.Button(root, text="Addition", command=lambda: arithmetic_operations("Addition"),width =20)
subtraction = tk.Button(root, text="Subtraction", command=lambda: arithmetic_operations("Subtraction"),width =20)
multiplication = tk.Button(root, text="Multiplication", command=lambda: arithmetic_operations("Multiplication"),width =20)
division = tk.Button(root, text="Division", command=lambda: arithmetic_operations("Division"),width =20)
addition.pack()
subtraction.pack()
multiplication.pack()
division.pack()
# Label created to display the output
output = tk.StringVar()
output_label = tk.Label(root,textvariable=output)
output_label.pack()
#mainloop() is a method and runs infinite loop.#mainloop() keeps the application running until the user closes the main window.
root.mainloop()
```

Figure 10.16: Program window of arithmetic operations performed using Tkinter

The output window is shown in Figure

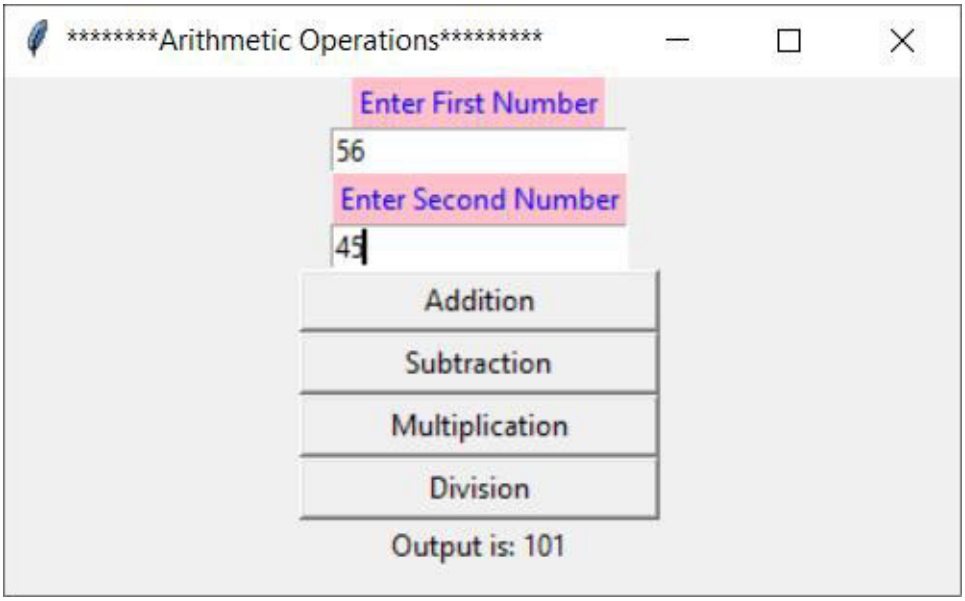


Figure 10.17: Output window of Arithmetic operations using Tkinter

Conclusion

In this chapter, we have covered the NumPy foundations for 1-dimensional array and matrix operations with the help of appropriate examples. In addition, the Tkinter library is introduced. Basic examples pertaining to the Tkinter main application window, label, button widget, and button click event, among other things, are covered.

In learners will get a quick recap of the practice exercises and multiple choice questions of Python.

Points to remember

NumPy, short for Numerical Python, is a well-liked open-source Python library for mathematical and numerical calculations.

Tkinter is a popular Python software for creating GUI-based desktop applications.

Tk() is a class, and an instance of this class is created to show the main application window.

The root is a variable for the main application window, that is, the root window.

pack() method is used to arrange the widgets.

mainloop() is a method that runs an infinite loop and keeps the GUI running until the user closes the main window.

Entry is also one of the widgets and is used for input fields.

The button is the widget used for firing events.

Exercises

What do you mean by NumPy?

Give an example of matrix addition and multiplication using NumPy.

What is Tkinter? Write the applications of Tkinter.

Discuss the program structure of Tkinter with the help of a suitable example.

Write a simple program of button click event in Tkinter.

Appendix

Practice Exercises with Solutions

Sample programs

Write a program for tuples using a for loop.

In this example, a tuple is created, and the elements are accessed using a for loop.

Code Appendix.1:

learners = "MCA", 89),("Ankit", "Btech", 97))

print('The list of learners is mentioned below')

for i in learners:

    print(i)

The program window is as shown in Figure

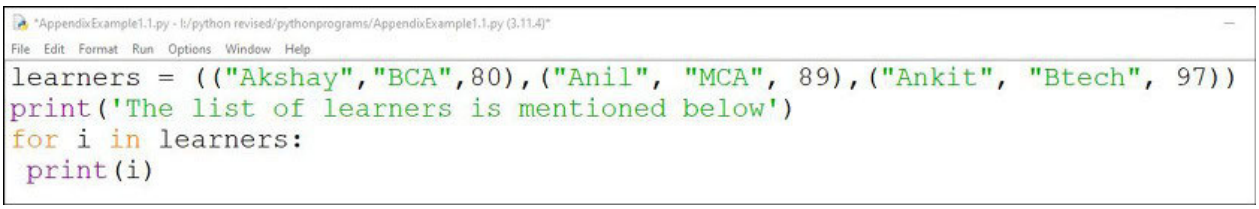

AppendixExample1.1.py - I:/python revised/pythonprograms/AppendixExample1.1.py (3.11.4)
File Edit Format Run Options Window Help
learners = (("Akshay","BCA",80),("Anil", "MCA", 89),("Ankit", "Btech", 97))
print('The list of learners is mentioned below')
for i in learners:
print(i)

Figure of tuples

The output of the program is shown in Figure

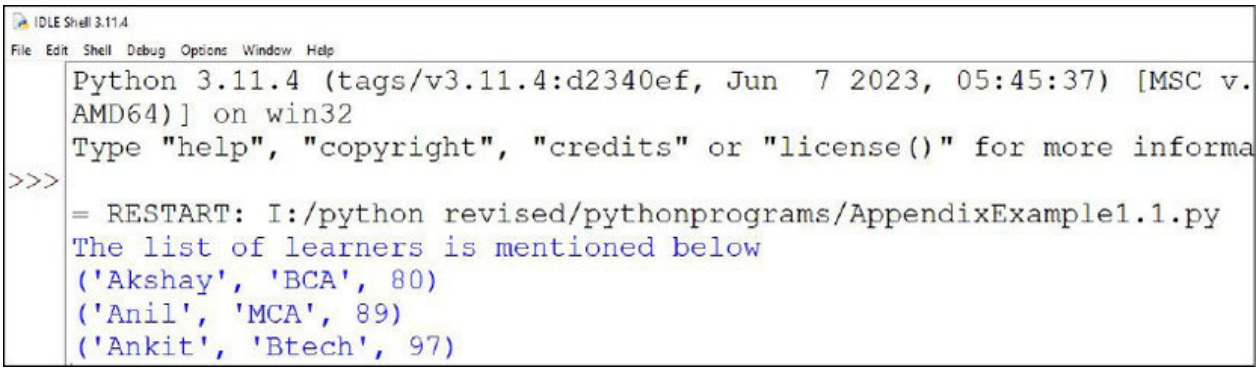

IDLE Shell 3.11.4
File Edit Shell Debug Options Window Help
Python 3.11.4 (tags/v3.11.4:d2340ef, Jun 7 2023, 05:45:37) [MSC v.
AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more informa
>>>
= RESTART: I:/python revised/pythonprograms/AppendixExample1.1.py
The list of learners is mentioned below
('Akshay', 'BCA', 80)
('Anil', 'MCA', 89)
('Ankit', 'Btech', 97)

Figure of tuples

Write a program for finding maximum and minimum values using functions.

Code Appendix.2:

```
def findmaxmin(data):
    x = max(data)
    y = min(data)
    return(x,y)
data = (19, 98, 90, 9)
(maximum, minimum) = findmaxmin (data)
print("Maximum Marks = ", maximum)
```

print("Minimum Marks = ", minimum)

In this example, a function named findmaxmin() is defined, and the parameter is passed into it. Here, the inbuilt max and min functions are used to find maximum and minimum values from the tuples defined. The program window is shown in Figure

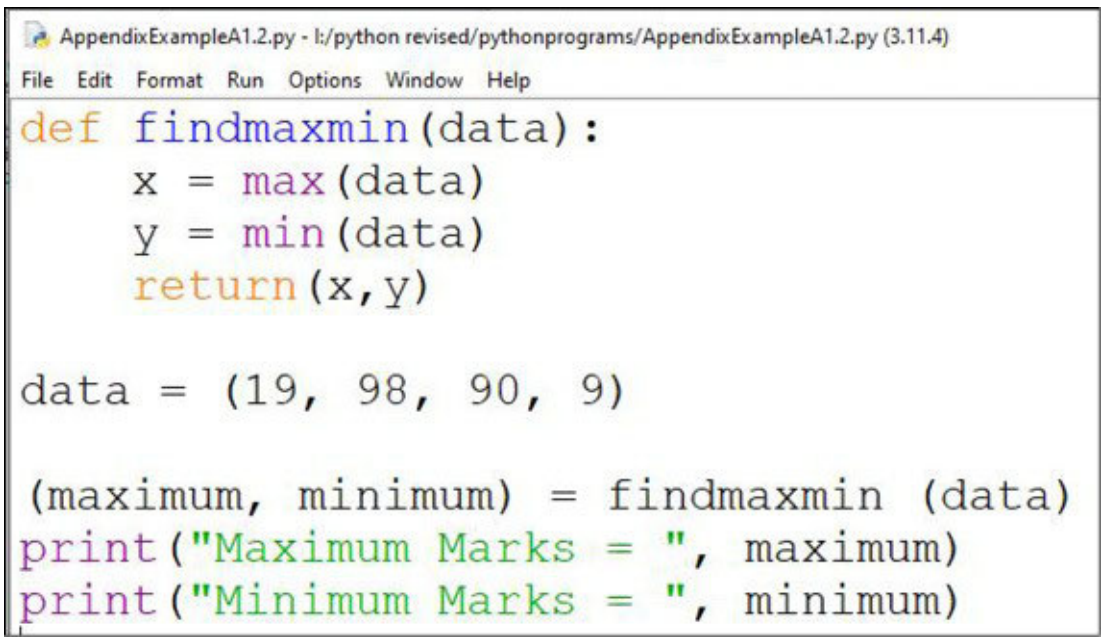

AppendixExampleA1.2.py - I:/python revised/pythonprograms/AppendixExampleA1.2.py (3.11.4)
File Edit Format Run Options Window Help
def findmaxmin(data):
x = max(data)
y = min(data)
return(x,y)
data = (19, 98, 90, 9)
(maximum, minimum) = findmaxmin (data)
print("Maximum Marks = ", maximum)
print("Minimum Marks = ", minimum)

Figure maximum and minimum values using function

The output of the program is shown in Figure

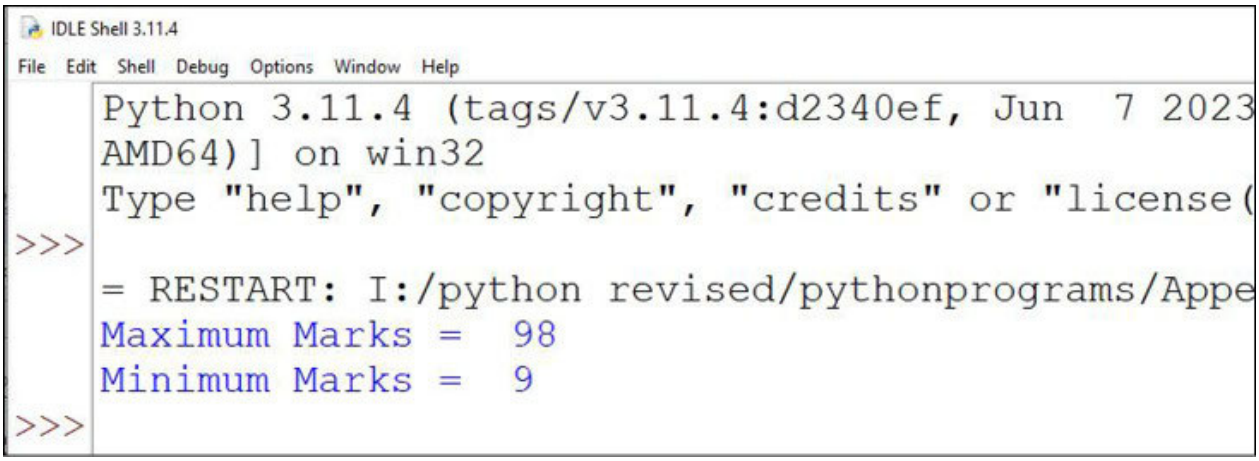


Figure of finding maximum and minimum value using function

Write a program for string concatenation.

In this example, strings are concatenated using a + symbol.

Code Appendix.3:

```
str1 = "Welcome in"

str2 = "PythonWorld"



str3 = str1 + str2

print ("the concatenated string is:",str3)
```

The program window in IDLE is shown in Figure

```
File  Edit  Format  Run  Options  Window  Help
str1 = "Welcome in"
str2 = "PythonWorld"
str3 = str1 + str2
print ("the concatenated string is:",str3)
```

Figure of string concatenation

The output of the program is shown in Figure

```
>>>
>>>
======= RESTART: I:/python revised/pythonprograms/AppendixExampleA1.3.py
the concatenated string is: Welcome inPythonWorld
```

Figure of string concatenation

Write a program to display firstname, lastname and student id of student using a class.

In this example, a class A is defined and y is the object of a class. On creating an object of a class, the values are passed as parameters in __init__() function and values are returned from __str()__ function.

Code Appendix.4:

```
class A:

    def __init__(self,fname,lname,sid):

        self.first = fname
```

self.last = lname

self.sno = sid

def __str__(self):

return self.first + " " + self.last + " " + self.sno

y = A("Sunil", "Kumar", "110")

print(y)

The program window is shown in Figure

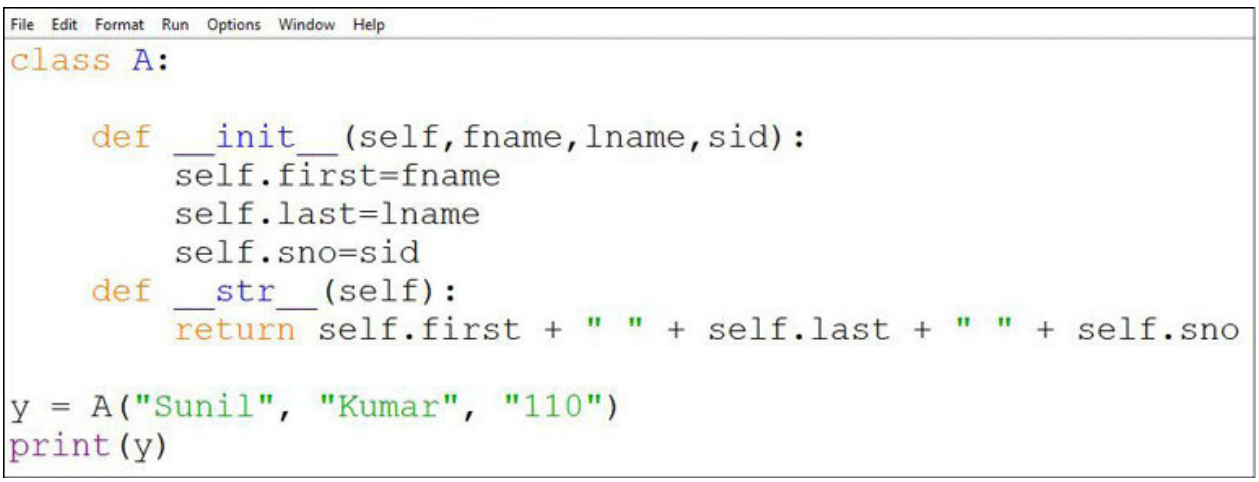


Figure of a class

The output of the program is shown in Figure

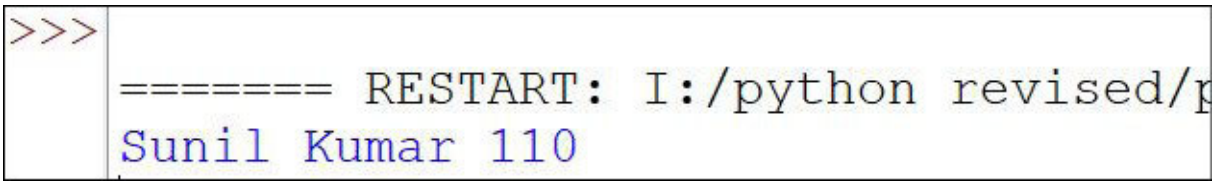


Figure of a class

Write a program to display firstname, lastname and rollno of student using inheritance.

In this example, inheritance is achieved in class B. The function Fullname() is defined in class A and is called in a function GetDetail() to display the firstname, lastname and the rollno. of a student. The variables used are: last and The object of the class A is x and the object of class B is

Code Appendix.5:

```
class A:

    def __init__(self,fname,lname):

        self.first = fname

        self.last = lname

    def Fullname(self):

        return self.first + "" + self.last
```

```
class B(A):

    def __init__(self,fname,lname,sid):

# the class 'A' variables are inherited in class B

        A.__init__(self,fname,lname)

        self.sno = sid

    def GetDetail(self):

        return self.Fullname() + "," + self.sno

x = A("Sunil","Kumar")

y = B("Anil","Kumar","429")

print(x.Fullname())



print(y.GetDetail())
```

The program window is shown in Figure

```
File  Edit  Format  Run  Options  Window  Help
class A:
#On creating objects of respective class,values are passed here __init__()
    def __init__(self,fname,lname):
        self.first = fname
        self.last = lname
# Function 'Fullname()' is defined here
    def Fullname(self):
        return self.first + "" + self.last
# class B is inheriting class A
class B(A):
    def __init__(self,fname,lname,sid):
# the class 'A' variables are inherited in class B
        A.__init__(self,fname,lname)
        self.sno = sid
# Function GetDetail() is defined here
    def GetDetail(self):
        return self.Fullname() + "," + self.sno
#Object x and y are created of class A and B
x = A("Sunil","Kumar")
y = B("Anil","Kumar","429")
#Functions are called using object x and y
print(x.Fullname())
print(y.GetDetail())
```

Figure of inheritance

The output of the program is shown in Figure

```
IDLE Shell 3.11.4
File  Edit  Shell  Debug  Options  Window  Help
    Python 3.11.4 (tags/v3.11.4:d2340ef, Jun  7 2023, 05:45:37) [MSC v.
    AMD64)] on win32
    Type "help", "copyright", "credits" or "license()" for more informa
>>>
    = RESTART: I:/python revised/pythonprograms/AppendixExampleA1.5.py
    SunilKumar
    AnilKumar,429
>>>
```

Figure of inheritance

Create a dictionary to display the following details in Python: {RollNo, Name, Course, CourseDuration}

In this example, a dictionary is created in the form of a {key: value} pair. A new value can be added in between.

Code Appendix.6:

list = {'RollNo': 1 ,'Name':'Ajay','Course':'BCA'}

print("list[RollNo] = " , list['RollNo'])

print("list[Name] = " , list['Name'])

print("list[Course] = " , list['Course'])

Add the value in between

list['CourseDuration']= "4 weeks"

print("list[CourseDuration] =", list['CourseDuration'])

The program window in IDLE is shown in Figure

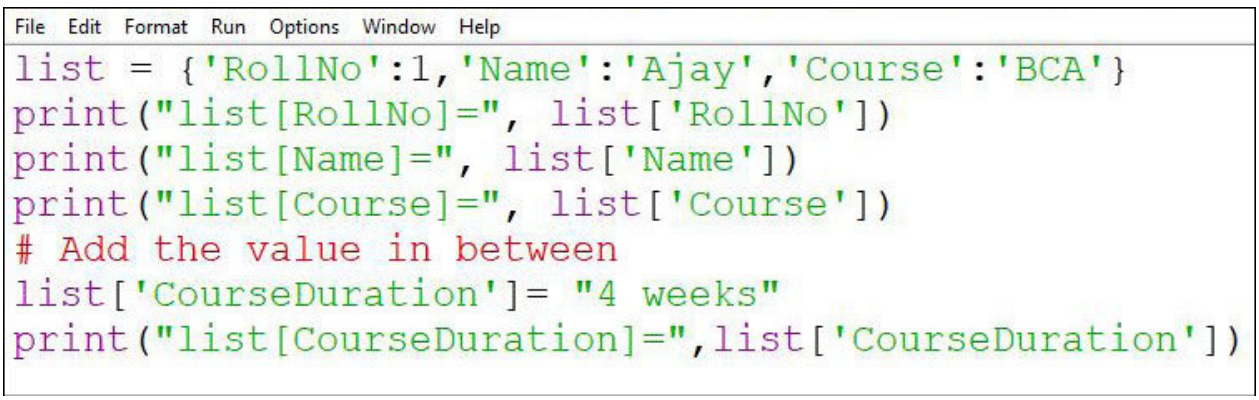


Figure of dictionaries

The output of the program is shown in Figure

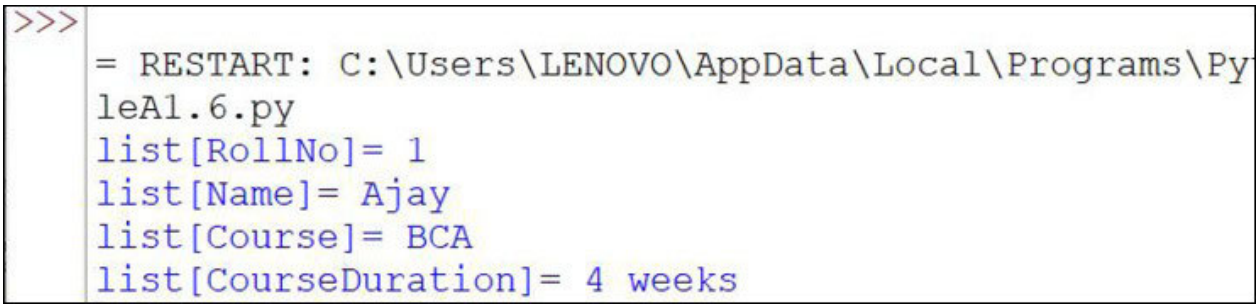


Figure of dictionaries

Write a program to find the length of an array.

In this example, an array is declared, and the length of an array is obtained using the len() function. The program window of the example is shown in Figure

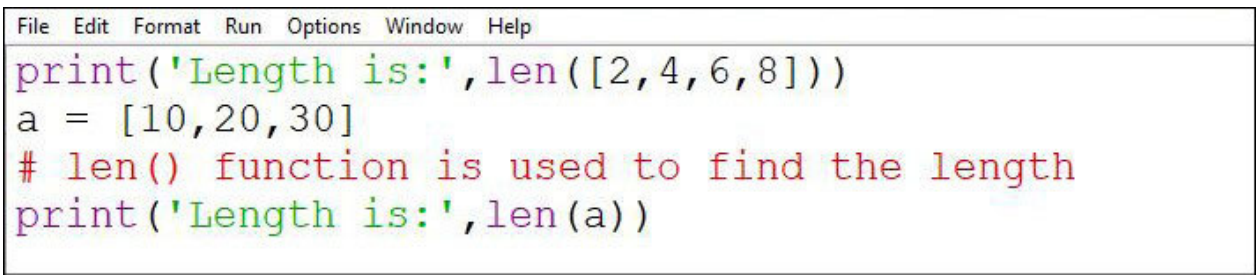

File Edit Format Run Options Window Help
print('Length is:',len([2,4,6,8]))
a = [10,20,30]
# len() function is used to find the length
print('Length is:',len(a))

Figure the length of an array

Code Appendix.7:

print('Length is:',len([2, 4, 6, 8]))

```
a = [10, 20, 30]

# len() function is used to find the length

print('Length is:',len(a))
```

The output of an example is shown in Figure

```
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Pro
leA1.7.py
Length is: 4
Length is: 3
```

Figure of example

Example of multiple if-else statements.

In this example, multiple if-else statements are used, and conditions are checked using relational operators.

Code Appendix.8:

```
# Input a number from the user

x = int(input("Enter a number: "))



# Use of Multiple If-Else Statements
```

if x > 0:

    print("The number is positive.")

elif x < 0:

    print("The number is negative.")

elif x==0:

    print("The number is zero.")

else:

    print("Input is wrong.")

The program window of example is shown in Figure

```
# Input a number from the user
x = int(input("Enter a number: "))
# Use of Multiple If-Else Statements
if x > 0:
    print("The number is positive.")
elif x < 0:
    print("The number is negative.")
elif x==0:
    print("The number is zero.")
else:
    print("Input is wrong.")
```

Figure of if-else statement

The output of the program is shown in Figure

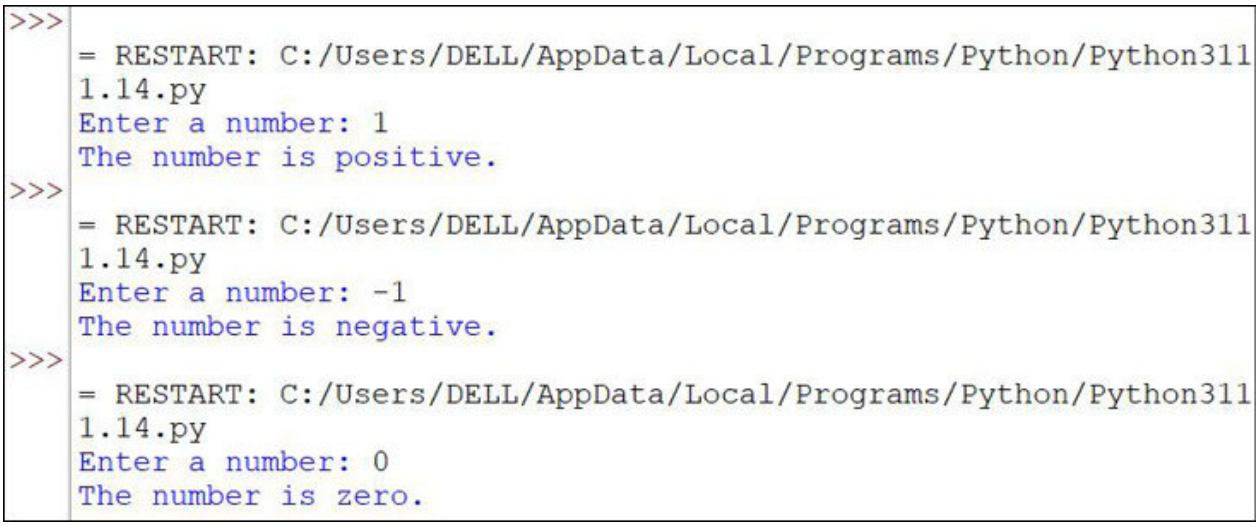


Figure of multiple if-else statements

Write a program to display the values from 1 to 10 using for loop.

Code Appendix.9:

```
for x in range(1, 11):

    print(x)
```

Explanation: The expression range (1, 11) assigns the variable n with the values 1, 2,..., 10. The syntax of the range function call is range (begin, end, step). The begin is the start value which is 1 in example, end is one value less than given that is 10 in example and step means the increment or decrement values. The program window of the example is shown in Figure

```
File  Edit  Format  Run  Options  Window  Help
for x in range(1, 11):
        print(x)
```

Figure of a range() function

The output of the example is shown in Figure

```
>>>
= RESTART: C:/Users/DELL/AppData/Local/
1.16.py
1
2
3
4
5
6
7
8
9
10
```

Figure of range() function and for loop

Write the program to find the square root of the number using inbuilt function:

In this example, a square root of a number is found using a predefined function and for that, an import statement is used to import predefined methods or functions of math. A program window of example is shown in Figure

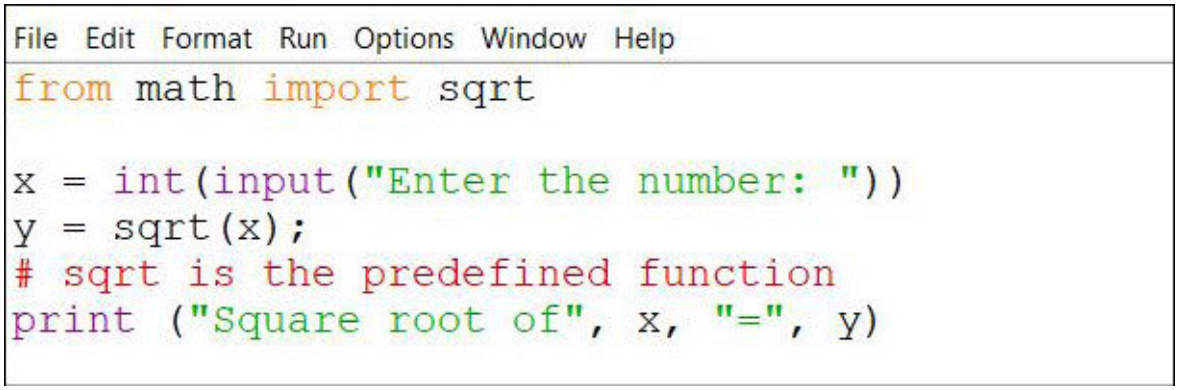

File Edit Format Run Options Window Help
from math import sqrt
x = int(input("Enter the number: "))
y = sqrt(x);
# sqrt is the predefined function
print ("Square root of", x, "=", y)

Figure of in-built function sqrt()

Code Appendix.10:

```
from math import sqrt

x = int(input ("Enter the number: "))

y = sqrt(x);

# sqrt is the predefined function

print ("Square root of", x, "=", y)
```

The output of the example is shown in Figure

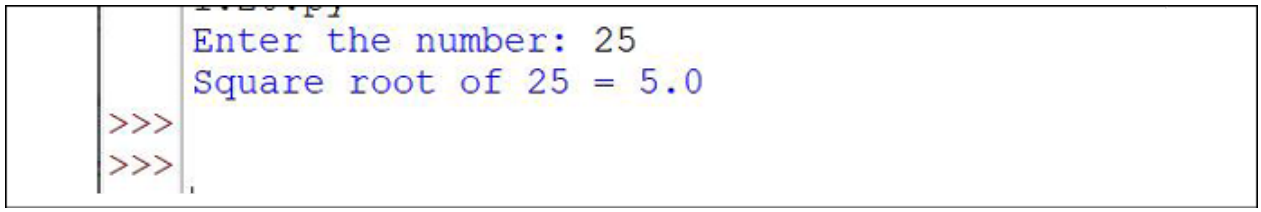


Figure window of in-built function sqrt()

Write the program to find the factorial using recursion.

Code Appendix.11:

```
def fact(x):

if x == 1:

    return 1

else:

    return (x * fact(x-1))

x = input("Enter the value")

#Here fact(x) function is called


print("The factorial of", x, "is", fact(x))
```

In this example, the concept of recursion is implemented. In recursion, the function calls itself. Here, the fact() function is defined and calls itself. The function stops only when the condition is satisfied. The program window of the example is shown in Figure

```
def fact(x):

        if x == 1:
            return 1
        else:
            return (x * fact(x-1))

x = int(input("Enter the value"))
#Here fact(x) function is called
print("The factorial of", x, "is", fact(x))
```

Figure of recursion

The output of the example is shown in Figure

```
Enter the value5
The factorial of 5 is 120
>>>
```

Figure of factorial using recursion

Example of parameter passing in functions

In this example, a function fun(x) is defined as a parameterized function. Another function defined in a program is Function fun(x) is called inside it, and the value of x=6 is passed as a parameter. When the value of x=6 is

passed inside a function fun(x) and is incremented by 1, it becomes 7 in the fun(x) function, but the value of x will remain 6 in the main() function. This is an example of callbyvalue() concept in which the actual argument value does not change, whereas the value is changed only inside a function like in the fun(x) function.

Code Appendix.12:

```
def fun(x):

    x += 1

    print("value of x in function=", x)

def main():

    x = 6

    fun(x)

    print("value of x in main function =", x)

# calling the main function to execute

main()
```

The program window is shown in Figure

```
def fun(x):
    x += 1
    print("value of x in function=", x)

def main():
    x = 6
# calling fun(x) function
    fun(x)
    print("value of x in main function =", x)
# calling the main function to execute

main()
```

Figure of parameter passing in a function

The output of the program is shown in Figure

```
value of x in function= 7
value of x in main function = 6
>>>
```

Figure of parameter passing in a function

Write a program to validate using exception handling that entered input value is integer value.

In this example, an input value is entered, and if the entered value is an integer, then no exception will be raised, and if the input value is not an integer, then an exception will be raised.

Code Appendix.13:

try:

n = int(input("Please enter integer value: "))

print("n=", n)

except ValueError:

print("The entered value is not integer value")

The program window is shown in Figure

```
try:
        n = int(input("Please enter integer value: "))
        print("n=", n)
except ValueError:
        print("The entered value is not integer value")
```

Figure of exception handling

The output of the program is shown in Figure

```
>>>
= RESTART: C:/Users/DELL/AppData/Local/Programs/Python/Python311/AppendixExample
1.31.py
Please enter integer value: 4
n= 4
>>>
= RESTART: C:/Users/DELL/AppData/Local/Programs/Python/Python311/AppendixExample
1.31.py
Please enter integer value: 4.5
The entered value is not integer value
>>>
= RESTART: C:/Users/DELL/AppData/Local/Programs/Python/Python311/AppendixExample
1.31.py
Please enter integer value: A
The entered value is not integer value
```

Figure of exception handling

Write a program for sorting numbers.

In this example, an array a is initialized with some elements. All the elements of an array are iterated using for loop from element to element. In order to sort elements of an array, nested for loop is applied and element at the element that is 12 is compared to the element that is 1 and so on using for loop. Here, if the element at the 0th position is greater than the element at the 1st position, then elements are swapped, and the iteration will continue for the next elements until all the elements are not sorted.

Code Appendix.14:

```
a = [12,1,3,23,5]

x = 0

print("Array elements before sorting")

for i in range(0,5):


    print(a[i])

for i in range(0,5):

  for j in range(i+1,5):

    if(a[i]>a[j]):
```

x=a[i]

a[i]=a[j]

a[j]=x

print("Elements after Sorting")

for i in range(0,5):

print(a[i])

The program window of example is shown in Figure

```
a = [12,1,3,23,5]
x = 0
print("Array elements before sorting")
for i in range(0,5):
   print(a[i])
for i in range(0,5):
   for j in range(i+1,5):
#Element at the 0th position is compared with the 1st element and so on.
      if(a[i]>a[j]):
       x=a[i]
       a[i]=a[j]
       a[j]=x
print("Elements after Sorting")
for i in range(0,5):
  print(a[i])
```

Figure of elements

The output of the program is shown in Figure

```
Array elements before sorting
12
1
3
23
5
Elements after Sorting
1
3
5
12
23
```

Figure of the sorted elements

Write a program to print the pyramid.

In this example, a pyramid is obtained as an output with the help of a nested for loop. In a nested for loop, an internal for loop works under the control of an external for loop. In the external for loop, for one step, the internal for loop iterates completely, and then again control goes to the external for loop. Further, for the next increment in the external for loop, the internal for loop will iterate completely, and so on.

Code Appendix.15:

```
for i in range(0,5):

  for j in range(0,i+1):

     print("* ",end= " ")  #end="" appends a space

  print("\n")
```

The program window of example is shown in Figure

```
for i in range(0,5):
    for j in range(0,i+1):
# end = " " appends a space
        print("* ",end= " ")

    print("\n")
```

Figure window of Pyramid example

The output of the program is shown in Figure

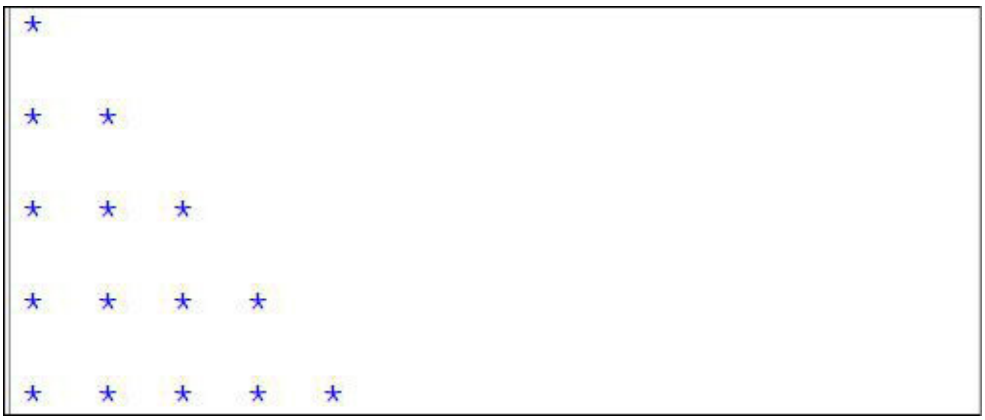


Figure of the pyramid example

Write a program for creating a tuple with different datatypes.

In this example, a tuple is created with 3 elements, and all elements are of different datatypes.

Code Appendix.16:

version = ("python", 3.7, 3)

print(version)

The program window is shown in Figure

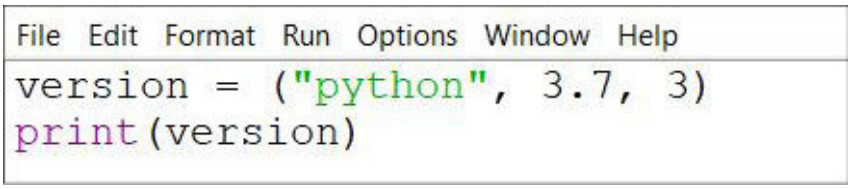


Figure a tuple of multiple datatypes

The output of the above example is shown below in Figure

Figure of tuple

Write the difference between lists, tuples and dictionary.

The difference between lists, tuples, and dictionary are mentioned in Table Appendix.1 below:

below:
below: below: below:

In this example, logical and or operators are discussed. In Logical both conditions should be true to get the output, whereas in Logical at least one condition should be true to get the output.

Code Appendix.17:

```
# Logical AND

# Returns True if both conditions are True

age = int(input("Enter your age"))

citizen = "India"

if age>=18 and citizen=="India":

    you are eligible for vote')

else:

    print('No,you are not eligible for vote')

# Logical OR



# Returns True if atleast one condition is True

income = int(input("Enter your income"))
```

if age>=18 or income>25000:

    print('Yes,you are eligible for taking Loan')

else:

    print('No,you are not eligible for taking loan')

The program window of this example is shown in Figure

```
# Logical AND
# Returns True if both conditions are True
age = int(input("Enter your age"))
citizen = "India"
if age>=18 and citizen=="India":
    print('Yes, you are eligible for vote')
else:
    print('No,you are not eligible for vote')
# Logical OR
# Returns True if atleast one condition is True
income = int(input("Enter your income"))
if age>=18 or income>25000:
    print('Yes,you are eligible for taking Loan')
else:
    print('No,you are not eligible for taking loan')
```

Figure of logical and, or operators

The output of this example is shown in Figure

```
>>>
    = RESTART: C:/Users/DELL/AppData/Local/Programs/Python/Python31
    A1.18.py
    Enter your age18
    Yes, you are eligible for vote
    Enter your income12000
    Yes,you are eligible for taking Loan
>>>
```

Figure window of logical and, or operators

Write a short note on NumPy.

The acronym for Numerical Python. It is the scientific computing core library. NumPy data structures are faster than lists and use less storage. Another useful application for NumPy is as an effective multi-dimensional generic data container. NumPy is well-known for arrays and matrices operations.

Write a program to show the concept of input() and format() functions.

In this example, the input() function is used to take input from the user, and the format() function is used with a {} placeholder to store the variable value and other expressions. f-format is a string format and is a shorter method of formatting strings.

Example displaying the input() and format() functions.

Code Appendix.18:

```
name = input("Enter Your Name:")
```

age = input("Enter Your Age:")

output = "Name is {} and Age is {} years old.".format(name, age)

print(output)

The above statement can also be written as in Code

result = f"Name is {name} and i am {age} years old."

#f symbolizes fstring format which offers a shorter method of formatting

#strings

print(result)

The program window is shown in Figure

```
#Example displaying the input() and format() functions
name = input("Enter Your Name")
age = input("Enter Your Age:")
output = "Name is {} and Age is {} years old.".format(name, age)
print(output)
#The above statement can also be written as:
result = f"Name is {name} and i am {age} years old."
#f symbolizes f-string format which offers a shorter method of formatting strings
print(result)
```

Figure of input() and format() functions

The output of above example is shown in Figure

```
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Py
leA20.py
Enter Your NameAjay
Enter Your Age:23
Name is Ajay and Age is 23 years old.
Name is Ajay and i am 23 years old.
```

Figure of input() and format() functions

Write a program using multiple if..elif..else statements.

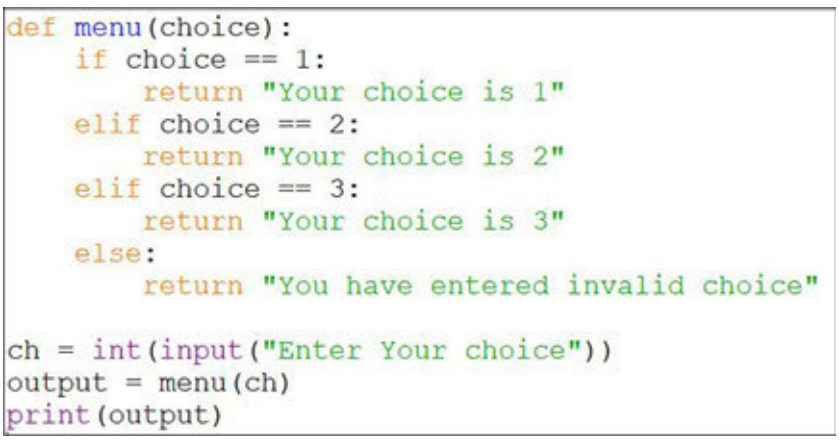
In this example, a function menu() is defined using the def keyword, and choice is taken as input from the user in the ch variable. The value is passed as a parameter to the function menu (choice), and using the if…elif...else statement, the choice is returned.

Code Appendix.19:

```
def menu(choice):
    if choice == 1:
        return "Your choice is 1"
    elif choice == 2:
        return "Your choice is 2"
    elif choice == 3:
```

```
        return "Your choice is 3"

    else:

        return "You have entered invalid choice"

ch = int(input("Enter Your choice"))

output = menu(ch)

print(output)
```

The program window is shown in Figure



```
def menu(choice):
    if choice == 1:
        return "Your choice is 1"
    elif choice == 2:
        return "Your choice is 2"
    elif choice == 3:
        return "Your choice is 3"
    else:
        return "You have entered invalid choice"

ch = int(input("Enter Your choice"))
output = menu(ch)
print(output)
```

Figure of if…..elif….else condition

The output of the example is shown in Figure

```
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python311
y
Enter Your choice2
Your choice is 2
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Python/Python311
y
Enter Your choice5
You have entered invalid choice
```

Figure window of if…elif…else condition

Write a program to find if the entered number is an Armstrong number or not.

In this program, a variable var1 contains the original number. In a while loop, the loop will continue until a number becomes 0. In this loop, a remainder is obtained using the modulus operator and the sum of digits, each raised to the power of the number of digits obtained in In the length of the input number entered is obtained. If the output in a variable result is equal to the original number entered, then the number is an Armstrong number. Else, it is not an Armstrong number. For example, 371 is an Armstrong number because 3^3+7^3+1^3 = 371 is obtained. Similar is the case for 153 because 1^3+5^3+3^3 = 153.

Code Appendix.20:

```
var1 = int(input("Enter a number to check"))

# length of a input number is obtained

var2 = len(str(var1))
```

```
# Initialize result variable to 0

result = 0

# Temporary variable to store the original number

temp = var1

# while loop to calculate the sum of digits

while temp > 0:

    var3 = temp%10

    result = result + var3**var2

    temp = temp//10

# Check if the result variable is equal to the original number

if var1==result:

    print(f"{var1} is an Armstrong number.")

else:



    print(f"{var1} is not an Armstrong number.")
```

The program window is shown in Figure

```
var1 = int(input("Enter a number to check"))
# length of a input number is obtained
var2 = len(str(var1))
# Initialize result variable to 0
result = 0
# Temporary variable to store the original number
temp = var1
# while loop to calculate the sum of digits

while temp > 0:
    var3 = temp%10
    result = result + var3**var2
    temp = temp//10

# Check if the result variable is equal to the original number
if var1==result:
    print(f"{var1} is an Armstrong number.")
else:
    print(f"{var1} is not an Armstrong number.")
```

Figure window of Armstrong number

The output window is shown in Figure

```
>>>
= RESTART: C:/Users/LENOVO/AppData/Local/Programs/Py
Enter a number to check371
371 is an Armstrong number.
>>>
=== RESTART: C:/Users/LENOVO/AppData/Local/Programs/
Enter a number to check153
153 is an Armstrong number.
```

Figure window of Armstrong number

Write a program to find the minimum and maximum elements from a list.

In this example, a list is defined, and a minimum and maximum element is obtained using the min() and max() prebuilt functions.

Code Appendix.21:

```
list1=[24,22,47,11,29,43]


# prebuilt max() and min() function to find smallest and largest number
if not list1:
    print("The list is empty.")
else:
minimum = min(list1)
    maximum = max(list1)
    print(f"The minimum element in the list is: {minimum}")
    print(f"The maximum element in the list is: {maximum}")
```

The program window is shown in Figure

```
list1=[24,22,47,11,29,43]

# prebuilt max() and min() function to find smallest and largest number
if not list1:
    print("The list is empty.")
else:
    minimum = min(list1)
    maximum = max(list1)
    print(f"The minimum element in the list is: {minimum}")
    print(f"The maximum element in the list is: {maximum}")
```

Figure Appendix.41: Finding maximum and minimum element from a list

The output of the example is shown in Figure

```
>>>
    = RESTART: C:\Users\LENOVO\AppData\Local\Programs\
    le1.24.py
    The minimum element in the list is: 11
    The maximum element in the list is: 47
```

Figure of maximum and minimum element

Output prediction

Problem 1

a = [1, 3, 6, 8]

print(a)

a = a + [2, 4, 5]

print(a)

[1,3,6,8]

[1,3,6,8]

[1,3,6,8]

[1,3,6,8,2,4,5]

[2,4,5]

[2,4,5,1,3,6,8]

None

b.

Problem 2

a = [1, 3, 6, 8]

print(a)

a + [2, 4, 5]

print(a)

[1,3,6,8] [1,3,6,8]

[1,3,6,8] [1,3,6,8,2,4,5]

[2,4,5] [2,4,5]

None

Answer: a.

Problem 3

```
a = [1, 3, 6, 8]

print(a)

a = a + [2, 4, 5]

print(a)

a += [10]

print(a)
```

[1,3,6,8]

[1,3,6,8,2,4,5]

[1,3,6,8,2,4,5,10]

[1,3,6,8]

[1,3,6,8,2,4,5]

[11,13,16,18,14,15]

[1,3,6,8]

[2,4,5]

[12,14,15]

None

a.

Problem 4

print(6)

print("6")

6

Error

6 6

None of these

c.

Problem 5

i = 7

print(i)

print("i")

7 i

7 7

i 7

None of these

Answer: a.

Problem 6

x1 = 1

x2 = 2

print(x1,x2)

if x1 == x2:

    print('Yes')

else:

```
    print('Not')
```

Yes (1,2)

(1,2) Not

Unexpected Indent

Error

b.

Problem 7

```
for x in range(5, 0, -2):


    print(x)
```

5 0 -2

5 3 0

5 4 3

5 3 1

d.

Problem 8: How many asterisks will be printed after running the below-mentioned statements?

```
for a in range(-5, 5):

print('*')
```

10

5

11

6

a.

Problem 9: What will be the output of the following program?

```
x = 6

while x > 0:

x = x-1
```

```
if x == 5:
    continue
print ('x value :', x)
```

5 4 3 2 1 0

4 3 2 1 0

6 4 3 2 1 0

5 4 3 2 1

b.

Problem 10

```
x = 0
n = 0
while(n<6):
    x = x + n
```

```
    n = n + 1
    if n==4:
        break
print("The output is:", x)
```

6

5

4

3

a.

Problem 11

```
def test(x):
    return x + 2
def main():
```

```
    x = test(3)

    print (x)

main()
```

a. 7

b. 6

c. 10

d. 5

d.

Problem 12

```
def fun(x, y):

   return 2*x + y*y

def main():


   print(fun(3,2))
```

main()

12

5

10

11

c.

Problem 13

```
def main():
    print(max(4, 5))
    print(max(5, 4))
    print(max(1, 2, 3))
main()
```

1 2 3

4 5

5 4

Error

d.

Problem 14

list = [1, 2, 3, 4, 5]

print(list)

list[3:4] = ['x', 'y']

print(list)

[1, 2, 3, 4, 5]

[1, 2, 3, ‘x’ , ‘y’ , 5]

[1, 3, 4, 5]

[1 , 3 , 4, ‘x’, ‘y’, 5]

[1 , 3 , 4]

[1, ‘x’, ‘y’, 5]

[1, 3, 2, 4 , 5]

[1,2,3,’x’,’y’,4]

a.

Problem 15

print("Hello" + "python")

Hello

python

pythonhello

Hellopython

d.

Problem 16

print ("defg"[2:])

f

g

fg

de

c.

Problem 17

var = "python example"

print("%s" %var[4:7])

th

ex

on

example

c.

Problem 18

x = (1,2)

print (2*x)

(1,2,1,2)

(1,1,1,2)

(2,1,2,1)

(1,2)

a.

Problem 19

var = "programming"

x = "gr"

while x in var:

    print(x, end= "")

Programming

gr

Infinite loop

None

c.

Problem 20

var = ['a', 'b', 'c' , 'd', 'e' , 'f']

print (var [5 :: -2])

[‘f’, ‘d’ , ‘b’]

[‘c’, ‘d’ , ‘b’]

[‘f’, ‘e’ , ‘d’]

[‘f’, ‘d’ , ‘c’]

a.

Hint: In print (var [5 :: -2]) statement, 5 means element at index 5 and it proceeds to the start of the list. The value is decremented by -2.

Problem 21

```
coursefee = {'C': 4000, 'java': 5000, 'dotnet': 6000}

print(coursefee['C'])
```

5000

4000

6000

None

b

Problem 22: What will be the output of the following:

```
coursefee = {}


coursefee['C'] = 4000

coursefee['java'] = 5000

coursefee['dotnet'] = 6000
```

print(coursefee)

{‘dotnet’:6000, ‘C’:4000, ‘java’:5000}

{‘C’:4000, ‘java’:5000, ‘dotnet’:6000}

{‘java’:5000, ‘C’:4000, ‘dotnet’:6000}

None

Hint: Empty dictionary is created and key-value pairs are assigned.

a

Problem 23

courselst = ["C++", "Python", "Dotnet"]

courselst.append("Android")

print(courselst)

[‘C++’, ‘Python’ , ‘Android’ , ‘Dotnet’]

[‘C++’, ‘Python’ , ‘Dotnet’ , ‘Android’]

[“C++”, “Python” , “Android” , “Dotnet”]

Error

b.

Problem 24

```
for x in range(1, 15, 2):
  print(x)
```

1,3,5,7,9,11,13

1,2,3,4,9,11,13

2,4,6,8,10,12,14

Error

a.

Problem 25: What will be the output of the following?

```
var1 = 8
def displayval():
```

```
    var1 = 12

print("value:", var1)

displayval();

print("value:", var1)
```

(‘value:’,12)

(‘value:’,12)

(‘value:’ ,12)

(‘value:’ , 8)

(‘value:’ , 8)

(‘value:’,12)

Error

Hint: Local variable scope is within the function only whereas the global variable scope is outside the function.

b.

Problem 26

a = 5

b = 1

c = a & b;

print "Value of c is", c

c = a | b;

print "Value of c is ", c

Value of c is 1

Value of c is 2

Value of c is 2

Value of c is 5

Value of c is 1

Value of c is 5

None

c.

Hint: Bitwise & and | operators are used here. c = a&b means binary of 5 and 1 that is 101 & 001. The condition in Bitwise AND (&) operator is that if both conditions are true, then output is true that is if 1 & 1 = 1, 1 & 0 = 0, 0 & 1 = 0, 0 & 0 = 0 whereas in Bitwise OR (|), if one condition is true and other is false, the output is true that is 1 | 0 = 1, 1|1 = 1, 0 |1 = 1, 0 | 0 = 0.

Problem 27

str = "programming"

print(str[2:5])

rog

ogr

rogr

rogra

b

Problem 28

```
print ((4, 2, 3) + (6, 5, 7))
```

(4, 2, 3) + (6 , 5, 7)

(4, 2, 3, 6, 5, 7)

(4, 2, 3, + 6, 5, 7)

Error

Answer: b

Problem 29

```
num = 1

def func():

    num = 3

    print(num)

func()

print(num)
```

1 3

3 1

the program has a runtime error because x is not defined.

1 1

Hint: Here the concept of Local and Global variables is implemented.

b

Problem 30

```
def func(x = 2, y = 3):
    x = x + y
    y += 1
print(x, y)
func(y = 3, x =2)
```

1 3

2 3

the program has a runtime error because x and y are not defined.

5 4

d

Problem 31

```
def func(msg, num = 1):
    print(msg * num)
func('Study')


func('Python', 4)
```

Study Python

Study PythonPythonPythonPython

Study Python, Python, Python

Study

b

Problem 32

```
def  func1(txt, n):
        while n > 0:
                print(txt)
                n = n - 1
func1('Python', 2)
```

Python

Python Python

invalid call

infinite loop

b

Problem 33

What will be the output of the following code?

```
student = {"name": "Sunil", "age": 20, "course": "Computer Science"}

print(len(student))
```

3

4

2

1

a.

Problem 34

What is the output of the following code?

```
tuple1 = (21, 32, 53, 44, 25)

output = tuple1[0:4]

print (output)
```

a. (21, 32)

b. (25, 32)

c. (21, 32, 53, 44)

d. (21, 32, 53, 44, 25)

c

Hint: In this example, a tuple is created and the values will be returned from 0th index to 3rd index only and the upper bound index that is 4th index value will be ignored.

Problem 35: After executing the following, output will be:

```
lst = [8, 3, 4, 6, 5, 7, 0, 2]

print (lst[:5])

print (lst[3:])

print (lst[-3:])

print (lst[::2])
```

[8, 3, 4, 6, 5]

[6, 5, 7, 0, 2]

[7, 0, 2]

[8, 4, 5, 0]

[3, 4, 6, 5]

[6, 5, 7, 0]

[7, 0, 2]

[8, 4, 5, 0]

[8, 3, 4, 6]

[5, 7, 0, 2]

[0, 2]

[4, 5, 0]

[8, 3, 4]

[6, 5, 7]

[7]

[8, 4]

a.

Hint: print (lst[:5]): In this statement, the list will execute from the 0th index position that is 8 and display the elements uptill the 4th index. The upper bound element that is 5 is neglected.

print (lst[3:]): Here, the list will start from the 3rd index that is 6 and will display elements up till the last.

print(lst[-3:]): In this statement, list execute from the end and 3rd index from last that is 7 and execute uptill 2.

print(lst[::2]): In this, ::2 is the extended slicing and start from 0th index and every second element will be displayed.

Problem 36

What will be the output of following?

var1 = 56

var2 = 79

var3 = var1+var2

var4 = var3 // 2

print (var4)

67

67.5

68

66.5

a.

Problem 37

var1 = 56

var2 = 79

var3 = var1+var2

var4 = var3/2

print (var4)

67

67.5

68

66.5

b.

Problem 38

var1 = 6

var2 = 7

var3 = var1+var2

var4 = var3 // 2

var5,var6 = var3,var4

print(str(var5) +"," + str(var6))

13,6

13,6.5

13,7

Error

a.

Problem 39

```
lst1 = [25,26,28,30]

print(lst1[1:3])

print(lst1[::2])

print(lst1[::-1])

print(lst1[::-2])


print(lst1[2::-1])

print(lst1[2::-2])

print(lst1[-2::1])
```

[26, 28][25, 28][30, 28, 26, 25][30, 26][28, 26, 25][28, 25][28,30]

[25, 28][25, 26][30, 28, 25, 26][30, 28][28, 26, 25][28, 25][28,26]

[25, 28][25, 26][30, 28, 25, 26][30, 28][28, 26, 25][28, 25][28,26]

[26, 28][26, 26][30, 28, 25, 26][30, 27][28, 25, 26][28, 26][28,30]

a.

Hint: In this example, in the first statement, print(lst1[1:3]) means to start from the 1st position, that is 26, because list starts from 0th index and prints elements uptill 2, therefore, the answer is [26,28].

In the second statement, print(lst1[::2]) means start from the 0th index and print every second element, therefore, answer is [25,28].

In third statement, print(lst1[::-1]) means print the elements in reverse order. The answer is: [30, 28, 26, 25].

In fourth statement, print(lst1[::-2]) means take the element from the end that is 30 and print every second element from the last, therefore, output is [30,26].

In the fifth statement, print(lst1[2::-1]), we start from the second element, that is 28, and then reverse it; therefore, the answer is [28,26,25]

In sixth statement, print(lst1[2::-2]), we start from 2nd element that is28 and then print the every second element in reverse, , the answer is [28,25].

In seventh statement print(lst1[-2::1]), we traverse from the end and take second element from last that is 28 is -2. Here, the element at -1 position is 30. Therefore, element at -2 position is 28 and print every first element, the answer comes as [28,30].

Problem 40

```
lst1 = [12,13,"Python",3.5]

print(lst1[1:3])

print(lst1[::2])

print(lst1[::-1])

print(lst1[::-2])

print(lst1[2::-1])

print(lst1[2::-2])
```

[13, ‘Python’]

[12, ‘Python’]

[3.5, ‘Python’, 13, 12]

[3.5, 13]

[‘Python’, 13, 12]

[‘Python’, 12]

[12, ‘Python’]

[12]

[3.5, ‘Python’, 13]

[3.5, 13]

[‘Python’, 13]

[‘Python’, 12]

[13, ‘Python’]

[12,’Python’]

[3.5, ‘Python’, 12]

[3.5, 13]

[‘Python’, 13]

[‘Python’, 12]

[13, ‘Python’]

[13,”Python”]

[3.5, ‘Python’, 12]

[3.5, 12]

[‘Python’, 13,12]

[‘Python’, 12]

a

Hint: For this example:

In the first statement, print(lst1[1:3]), we start from the first position and print elements uptill 2, therefore, answer is [13,’Python’].

In second statement, that is print(lst1[::2]), we start from the 0th index that is 12 and print every second element, therefore answer is [12,’Python’].

In third statement, that is print(lst1[::-1]), the elements are printed in a reverse order, therefore, the answer comes as [3.5,’Python’,13,12].

In fourth statement, that is print(lst1[::-2]) the element from the end, that is 3.5, and every 2nd last element is printed, therefore, output is [3.5,13].

In fifth statement, that is, print(lst1[2::-1]), we start from the second element, that is, ‘Python’ and then reverse, therefore, answer is [‘Python’,13,12].

In sixth statement, that is print (lst1[2::-2]), we start from the second element that is ‘Python’ and then print the every second element in reverse, therefore, the answer is [‘Python’,12].

Problem 41

```
for i in range(1,6):
    if i==3:
        continue
    print(i)
```

1 2 4 5

2 3 4 5

1 2 4 5 6

None

a

Hint: In this example, elements traverse from 1 to 5 and if the value of “i” matches to 3, then the continue statement will skip the current step, and print the other steps.

Problem 42: What will be the output of following?

```
for i in range(1,6):
    if i==3:
        break
    print(i)
```

1 2 3

1 2

1 2 3 4 5

None

b.

Hint: In this example, on encountering the break statement, loop will be stopped, therefore, the result is option (b).

Problem 43

```
# Tuple Packing

tup1=(21,34,"Python",78)

# Tuple Unpacking

var1,var2,var3,var4=tup1

print(var1,var2,var3,var4)
```

21,34,”Python”,78

34,21, “Python”, 78

21,34,78, “Python”

21,34, “Python”

a

Hint: In this example, Tuple packing and unpacking is done. The elements of tuple will be stored in the variables var1, var2, var3 and var4.

Problem 44: The output of statement >>>print(36/9) is:

4.0

4

error

9

a.

Problem The output of statement >>>print(36//9) is:

4

4.0

36

9.0

a.

Problem 46: What will be the

var1 = 10

var2 = var1= var1*5

print(var2)

50

10

5

Error

a

Hint: Here, concept of unpacking is used, the values are extracted back into variables.

Problem 47: What will be the output of following?

print(sum(a))

a. 57.5

b. 56.5

c. 55

d. 34.5

a.

Problem 48

```
import numpy

a1 = numpy.array([15,25,34,8,3,17])


var = numpy.max(a1)

print(var)
```

3

25

34

17

c.

Problem 49: What will be the output of the following slicing using list?

```
str1 = "Hello Python"

print(str1[-1])

print(str1[::-1])

print(str1[:-1])

print(str1[4:8])

print(str1[:2])

print(str1[3:])

print(str1[1:12:3])
```

n

nohtyPolleH

Hello Pytho

o Py

He

lo Python

o

nohtyP

Hello Pyth

o P

Hel

lo Python

not

nohtyPolleH

Hello Pytho

o Py

He

lo Python

noht

olleH

Hello

o Py

He

lo Python

a.

a. a. a.

Problem 50: What will be the output of the following statement?

** 3 ** 2)

27

19683

729

18

b

Hint: 3x3x3=27x27x27=19683

Problem 51: What will be the output of the following set statement?

>>>s1=set("HELLO PYTHON")

>>>s1

a. {‘L’, ‘P’, ‘N’, ‘O’, ‘E’, ‘ ‘, ‘Y’, ‘T’, ‘H’}

b. {‘H’, ‘E’, ‘L’, ‘L’, ‘O’, ‘ ‘, ‘P’,’Y’, ‘T’, ‘H’, ‘O’, ‘N’}

c. {‘L’, ‘P’, ‘N’, ‘O’, ‘E’, ‘ ‘, ‘Y’, ‘T’, ‘H’, ‘O’}

d. {‘L’, ‘P’, ‘N’, ‘O’, ‘E’, ‘ ‘, ‘Y’, ‘T’, ‘H’, ‘O’, ‘L’}

a

Hint: No duplicate elements appear in sets, and sets are mutable.. The order in sets is also undefined and sets can also consist of mixed values.

Problem 52: What will be the output of the following set statement?

>>>s2 = set ([1,1,2,'PYTHON','HELLO',1,'HELLO'])

>>>s2

{‘PYTHON’, 1, 2, ‘HELLO’}

{‘PYTHON’, ‘PYTHON’, 1, 2, ‘HELLO’}

{‘PYTHON’, 1, 2, ‘HELLO’, ‘HELLO’}

{‘PYTHON’, 1, 2, 2, 1, ‘PYTHON’, ‘HELLO’}

a.

Problem 53

lst1=[32,23,54,7,32,6]

lst1.pop(2)

print(lst1)

[32, 23, 7, 32, 6]

[32, 23, 54, 7, 32, 6]

[32, 23, 54, 32, 6]

[32, 7, 54, 32, 6]

a

Hint: The pop() method removes the element at the second 2nd position that is 54.

Problem 54

```
def average(x,y,z):
    avg= (x+y+z)/3
    return avg
output = average(5,3,4)
```

print("Avg of numbers is:",output)

4

4.0

12

None

b

Hint: In this example, a function is defined and the values are passed as parameter in a function average() and a value is returned after taking the average of three numbers.

Problem 55

```
def si(p,r=3,t=2):

interest= (p*r*t)//100

    print(interest)

si(100)

si(100,4)
```

si(100,3,3)

6 8 9

6.0 8.0 9.0

6.0 8.0 9

Error

a

Hint: In this example, a function si() is defined using a ‘def’ keyword. In first statement si(100), 100 is passed as a parameter in function si() and using simple interest formula, output is obtained. Similar is the case for the other two statements. Floor division (//) operator is used in example to obtain the output without decimals. Therefore, result is 6 8 9.

Multiple choice questions

In Python dictionary:

Items are accessed by their position in a dictionary

Dictionaries are mutable and can be nested

Dictionary contains elements of same data type

Dictionary and lists are same

b.

def is:

used to define function in python

a keyword in python

start of function header

all of the above

d

Which operator has the highest precedence?

()

**

|

&

a

w mode in file handling:

opens a file for writing only

overwrites the file if the file exists

creates a new file for writing, if the file doesn’t exist

All of the above

d

Fill the missing terms:

print ("Your name is _____ and Age is _____" % ('Anil', 22))

%s, %d

%d , %s

%d , %d

%s , %f

a

Which function header syntax is correct?

def f(a = 1, b):

def f(a = 1, b, c = 2):

def f(a = 1, b = 1, c = 2):

def f(a = 1, b = 1, c = 2, d):

c.

Which of the following is an advantage of using modules?

It provides a means of reuse of program code.

It provides a means of dividing up tasks.

It provides a means of reducing the size of the program.

It provides a means of testing individual parts of the program.

Choose the right option

Only a

a and b

a, b, d

a, b, c, d

c

Note: With the use of modules, you can divide a program into more manageable, smaller chunks. It allows for reusing code in various program sections or across multiple applications, which facilitates a more effective and efficient approach to development.

Which of the following is a correct way to declare a variable in Python?

xyz = “value”

xyz := “value”

“xyz” = var_name

$xyz = “value”

a

In Python, what does the len() function do?

Returns the sum of all elements in a list.

Returns the largest element in a list.

Returns the number of elements in a list.

Returns the average of all elements in a list.

c

What is the scope of a global variable in Python?

It is accessible only within the function where it is defined.

It is accessible from any function or block of code in the entire program.

It is accessible only within the module where it is defined.

It is accessible only within a class where it is defined.

b

____method tells the current position within the file.

tell()

fseek()

fopen()

none

a

_____method is used to set the files current position at the offset.

seek()

tell()

read()

write()

a.

_____ function can be used to check the data type of the variables.

ctype()

type()

typedef()

int

b.

Fill in the blanks

_____function is used to iterate through all the elements of an iterator.

Answer: next ()

_____mode opens a file for reading only.

Answer: r

The _____function returns an iterator that yields a sequence of evenly spaced integers.

Answer: range ()

_____keyword is used to define a function.

Answer: def

In functions, _____statement means that the value is to be returned to the caller.

Answer: return

If the variable is _____, then it can be accessed from anywhere in the program.

Answer: Global

Compared to tuples, _____are variable-length and their contents can be modified.

Answer: Lists

_____notation is used to select sections of most sequence types,the basic form consists of start: stop.

Answer: slice

_____ is a pre-defined python data structure and is commonly named as associative array.

Answer: dictionary

_____method is used to add a new element to a list.

Answer: append()

_____method is used to delete elements from a list.

Answer: pop ()

True/False

The word elif is a contraction of else and if.

Answer: True

A list refers to a collection of objects and represents an ordered sequence of data.

Answer: True

Lists are mutable and tuples are immutable.

Answer: True

The dot (.) operator is used to access elements of an object.

Answer: True

Python is a case-sensitive language.

Answer: True

The tuples are immutable and don’t have an append () method.

Answer: True

“in” is a keyword in Python.

Answer: True

A list is an unordered collection of objects and may contain any number of elements.

Answer: False

Import statement imports the module if the module is present in the search path, that is, the directory.

Answer: True

A tuple is a fixed-length, immutable sequence of Python objects.

Answer: True

Tuples can contain elements of different data types.

Answer: True

Dictionaries are mutable, and elements can be changed after creation.

Answer: True

A substitution for a value used as input by a function is known as a parameter in a function.

Answer: True

The primary purpose of a function in Python is to perform a specific task or computation.

Answer: True

Two lists are concatenated together with ‘+’ sign.

Answer: True

Negative indices slice the sequence from the starting.

Answer: False

Dictionary is a collection of key-value pairs, where key and value are Python objects.

Answer: True

1one_var is the correct way of writing a variable.

Answer: False

readlines() method returns a list.

Answer: True

‘r’ and ‘a’ modes represent read and append.

Answer: True

The print is a built-in function and prints a value directly to the system output.

Answer: True

Indexing is not valid in dictionary data type.

Answer: True

A set is an ordered collection of data types that is iterable, mutable, and has no duplicate elements.

Answer: True

Sets can be created by using the built-in set () function.

Answer: True

append () method adds a new element to the end of a list.

Answer: True

Join our book's Discord space

Join the book's Discord Workspace for Latest updates, Offers, Tech happenings around the world, New Release and Sessions with the Authors:

https://discord.bpbonline.com

Index

Symbols

__init__() function 120

A

Anaconda 5

arithmetic operators 14

arrays 52

append() function 56

for loop, using 56

in operator and if condition example 54

len() function example 55

membership operator example 53

program, writing with append() function 53

assignment operator 17

example 20

B

bitwise operators 17

bitwise left shift operator 20

bitwise right shift operator 20

example

break and continue statements

C

calendar module

using 142

classes 120

example 174

program, writing for 121

program, writing for functions 122

coursemodule1.py file 93

D

databases 102

data control language 103

data definition language 102

data manipulation language 103

data query language 102

program, writing for database and table creation

program, writing for deleting data 116

program, writing for updating data 114

records, reading and displaying from table

transaction control language 104

data control language (DCL) 103

data definition language (DDL) 102

data manipulation language (DML) 103

data query language (DQL) 102

date and time functions 140

program, writing to display datetime 141

program, writing to display time 141

program, writing to obtain calendar month 142

dictionaries 84

example 176

nested dictionary 87

program, writing for 84

values, accessing with index and arrays 86

E

encapsulation 128

program, writing for 129

public access specifier, for writing program 130

exception 146

exception handling 145

assert statement example 152

example 182

program, for division by zero exception 151

program, for error occurred due to inappropriate values 147

program, for error raised due to different data types 148

program of IOError exception 149

program of ModuleNotFound exception

F

fetchall() method 111

file handling

append mode, illustrating 101

data writing into file, with w mode 100

every line, reading in file 99

file, reading 99

files and modes 98

findall() function 138

for loop

example

format() function 189

function 57

local and global variables 58

parameters, passing 58

G

generators 78

example 80

programs, writing for

getdata() method 126

getTotal() function 126

Graphical User Interface (GUI) toolkit 160

I

identity operators 20

is identity operator 20

isnot identity operator 20

output 21

IDLE 3

IDLE Python 3.11 Shell 4

if conditions 28

example 29

example of checking, with and operator 31

example of checking, with or operator 31

example, using and, or operators 30

inheritance 123

example 175

program, writing for 124

input() function 189

IOError exception 148

iterators 76

example 77

using 78

L

lambda function 65

using 66

len() function 55

lists 70

append() function, using 71

local and global variables 59

example

logical operators 17

example

loops 33

for loop example

while loop example

M

membership operators 21

precedence 23

using 22

Microsoft SQL Server 102

mobile_validate() function 139

ModuleNotFound exception 149

modules 90

examples of using dictionary

mathematical functions, importing 94

using 92

multiple if-else statements

example 178

MySQL 102

MySQL Connector 108

N

nested dictionary 87

example

nested if condition 31

example 32

NoSQL databases 102

NumPy 155

1D-array operations 156

basic operations 158

matrix addition and multiplication 158

maximum, finding 157

mean, finding 157

minimum, finding 157

program, for creating 1D-array from Python list 157

program, for matrix addition and multiplication 159

sum, finding 157

O

object-oriented programming 120

classes 120

encapsulation 127

inheritance 123

objects 120

overriding 124

objects 120

program, writing for 121

operators, in Python 13

arithmetic operators 14

assignment operator 17

Bitwise operators 17

identity operators 20

logical operators 17

membership operators 21

relational operators

Oracle Database 102

overriding 125

program, writing for 126

program, writing for inheritance and function overriding 127

P

parameter passing

example 182

pop() method in dictionary

example 90

PostgreSQL 102

program

writing, for creating tuple 186

writing, for finding array length 177

writing, for finding factorial 181

writing for finding if entered number is Armstrong 192

writing for finding max and min values using functions 172

writing for finding minimum and maximum elements from list 193

writing, for printing pyramid 185

writing, for sorting numbers 184

writing for string concatenation 174

writing using logical and or operators 189

writing using multiple if..elif..else statements 191

PyCharm 5

Python 1

addition of two numbers example 10

dictionaries 84

exploring 8

installation

operators 13

program to print Hello World 8

program to swap two values 11

type() function 23

user input example 10

variable declaration 9

Python datatypes 12

type conversion 13

R

range() function 55

example 179

recursion

example 180

regular expressions 134

mobile number digits, verifying 140

program to check email ID pattern 137

program to search string 135

program, writing for substring function 134

program, writing for validating numerical values only 139

program, writing for validating string and values only 138

relational database management systems (RDBMS) 102

relational operators

re module 134

S

sets 81

example 82

slicing 71

example

square root 179

T

Tkinter 160

application window example 160

button click event example 164

input from user, using text field

label and button widget 161

label and button widget, creating 162

program for performing arithmetic operations

transaction control language (TCL) 102

commands, using

tuples 75

concatenation 75

nested tuples 76

program, writing using for loop 171

type() function 23

program, writing for 24

V

Visual Studio Code (VS Code) 5

W

WHERE clause 115

while loop

example